Task 1:

Create a node  for a tree and include a constructor.

public class Node {

int value;

Node left, right;

public Node(int value) {

this.value = value;

this.left = null;

this.right = null;

}

}

public class Main {

public static void main(String[] args) {

// Creating nodes

Node node1 = new Node(10); // Node with value 10

Node node2 = new Node(20); // Node with value 20

Node node3 = new Node(30); // Node with value 30

// Linking nodes to form a tree

node1.left = node2; // node1's left child is node2

node1.right = node3; // node1's right child is node3

// Example output

System.out.println("Root value: " + node1.value); // Should print 10

System.out.println("Left child value: " + node1.left.value); // Should print 20

System.out.println("Right child value: " + node1.right.value); // Should print 30

}

}

Output :

10

/ \

20 30

Task 2:

Create a class named Binarty Search tree in which you have 2 insert operations

1 insert —----> for inserting if the tree is empty

1 insert —----> for inserting if the tree has 1 or more nodes

public class BinarySearchTree {

class Node {

int value;

Node left, right;

public Node(int value) {

this.value = value;

this.left = this.right = null;

}

}

// Root of the BST

private Node root;

public BinarySearchTree() {

this.root = null;

}

public void insert(int value) {

if (root == null) {

root = new Node(value);

System.out.println(value + " inserted as the root node.");

} else {

insertRec(root, value);

}

}

private void insertRec(Node root, int value) {

if (value < root.value) {

if (root.left == null) {

root.left = new Node(value);

System.out.println(value + " inserted as left child of " + root.value);

} else {

insertRec(root.left, value);

}

} else if (value > root.value) {

if (root.right == null) {

root.right = new Node(value);

System.out.println(value + " inserted as right child of " + root.value);

} else {

insertRec(root.right, value);

}

}

}

public void inorder() {

inorderRec(root);

}

private void inorderRec(Node root) {

if (root != null) {

inorderRec(root.left);

System.out.print(root.value + " ");

inorderRec(root.right);

}

}

public static void main(String[] args) {

BinarySearchTree bst = new BinarySearchTree();

bst.insert(10);

bst.insert(5);

bst.insert(15);

bst.insert(12);

bst.insert(18);

System.out.print("Inorder traversal of the tree: ");

bst.inorder(); // Should print values in sorted order

}

}

Output:

10 inserted as the root node.

5 inserted as left child of 10

15 inserted as right child of 10

12 inserted as left child of 15

18 inserted as right child of 15

Inorder traversal of the tree: 5 10 12 15 18

Task 3:

Ionorder travel of the above code snippets from task 1 and Task 2

public class Node {

int value;

Node left, right;

// Constructor to initialize the node with a value

public Node(int value) {

this.value = value;

this.left = null;

this.right = null;

}

// Inorder traversal (left, root, right)

public void inorderTraversal() {

if (left != null) {

left.inorderTraversal(); // Visit left subtree

}

System.out.print(value + " "); // Visit root node

if (right != null) {

right.inorderTraversal(); // Visit right subtree

}

}

public static void main(String[] args) {

// Create nodes for the tree

Node root = new Node(10);

root.left = new Node(5);

root.right = new Node(15);

root.left.left = new Node(3);

root.left.right = new Node(7);

root.right.left = new Node(12);

root.right.right = new Node(20);

// Inorder Traversal

System.out.print("Inorder traversal of the tree: ");

root.inorderTraversal(); // This should print nodes in sorted order

}

}

Output:

Inorder traversal of the tree: 3 5 7 10 12 15 20

public class BinarySearchTree {

// Definition of the Node class

class Node {

int value;

Node left, right;

public Node(int value) {

this.value = value;

this.left = this.right = null;

}

}

// Root of the BST

private Node root;

// Constructor for BinarySearchTree

public BinarySearchTree() {

this.root = null;

}

// Insert method for an empty tree

public void insert(int value) {

if (root == null) {

// Tree is empty, create the root node

root = new Node(value);

System.out.println(value + " inserted as the root node.");

} else {

// Tree is not empty, insert using recursive method

insertRec(root, value);

}

}

// Recursive method to insert nodes into the BST

private void insertRec(Node root, int value) {

if (value < root.value) {

// Insert into the left subtree

if (root.left == null) {

root.left = new Node(value);

System.out.println(value + " inserted as left child of " + root.value);

} else {

insertRec(root.left, value);

}

} else if (value > root.value) {

// Insert into the right subtree

if (root.right == null) {

root.right = new Node(value);

System.out.println(value + " inserted as right child of " + root.value);

} else {

insertRec(root.right, value);

}

}

}

// Inorder Traversal of BST

public void inorder() {

inorderRec(root);

}

// Recursive method for inorder traversal

private void inorderRec(Node root) {

if (root != null) {

inorderRec(root.left); // Visit left subtree

System.out.print(root.value + " "); // Visit root node

inorderRec(root.right); // Visit right subtree

}

}

// Main method for testing

public static void main(String[] args) {

BinarySearchTree bst = new BinarySearchTree();

// Insert nodes into the BST

bst.insert(10);

bst.insert(5);

bst.insert(15);

bst.insert(3);

bst.insert(7);

bst.insert(12);

bst.insert(20);

// Inorder Traversal of the BST

System.out.print("Inorder traversal of the BST: ");

bst.inorder(); // This should print nodes in sorted order

}

}

Inorder traversal of the BST: 3 5 7 10 12 15 20

Task 4:

Create  a main method Task 1, 2 and 3

And run the code.

public class Node {

int value;

Node left, right;

public Node(int value) {

this.value = value;

this.left = null;

this.right = null;

}

public void inorderTraversal() {

if (left != null) {

left.inorderTraversal();

}

System.out.print(value + " ");

if (right != null) {

right.inorderTraversal();

}

}

public static void main(String[] args) {

Node root = new Node(10);

root.left = new Node(5);

root.right = new Node(15);

root.left.left = new Node(3);

root.left.right = new Node(7);

root.right.left = new Node(12);

root.right.right = new Node(20);

System.out.print("Inorder traversal of the tree: ");

root.inorderTraversal();

System.out.println();

BinarySearchTree bst = new BinarySearchTree();

bst.insert(10);

bst.insert(5);

bst.insert(15);

bst.insert(3);

bst.insert(7);

bst.insert(12);

bst.insert(20);

System.out.print("Inorder traversal of the BST: ");

bst.inorder();

}

}

class BinarySearchTree {

class Node {

int value;

Node left, right;

public Node(int value) {

this.value = value;

this.left = this.right = null;

}

}

private Node root;

public BinarySearchTree() {

this.root = null;

}

public void insert(int value) {

if (root == null) {

root = new Node(value);

System.out.println(value + " inserted as the root node.");

} else {

insertRec(root, value);

}

}

private void insertRec(Node root, int value) {

if (value < root.value) {

if (root.left == null) {

root.left = new Node(value);

System.out.println(value + " inserted as left child of " + root.value);

} else {

insertRec(root.left, value);

}

} else if (value > root.value) {

if (root.right == null) {

root.right = new Node(value);

System.out.println(value + " inserted as right child of " + root.value);

} else {

insertRec(root.right, value);

}

}

}

public void inorder() {

inorderRec(root);

}

private void inorderRec(Node root) {

if (root != null) {

inorderRec(root.left);

System.out.print(root.value + " ");

inorderRec(root.right);

}

}

}

Output:

Inorder traversal of the tree: 3 5 7 10 12 15 20

Inorder traversal of the BST: 3 5 7 10 12 15 20

Task 5:

Applications of Trees

**1. Hierarchical Data Representation**

* **File Systems**: Operating systems use trees to represent files and directories. The file system structure is inherently hierarchical, where directories are nodes and files are leaves.
  + **Example**: A file system on a computer can be modeled as a tree where the root is the root directory, and every subdirectory and file is a node or leaf in the tree.
* **XML/HTML Documents**: XML and HTML documents are often represented as trees, where each element or tag is a node.
  + **Example**: In an HTML document, the <!DOCTYPE> is the root node, <html> is the next, followed by <head> and <body>, and the text or elements inside these tags are children of these nodes.

**2. Binary Search Trees (BST) for Searching and Sorting**

* **Efficient Searching and Sorting**: Binary search trees are used for efficient searching, insertion, and deletion of elements. The average time complexity for searching in a BST is O(log N), making it ideal for dynamic searching.
  + **Example**: A **dictionary** where words are stored in a tree structure allows fast lookups and insertions.
  + **Applications**: Implementing associative arrays, sets, and maps.

**3. Database Indexing**

* **B-Trees and B+ Trees**: These are specialized tree structures used in databases to store indexes. They are highly balanced and optimized for systems that read and write large blocks of data (like disk storage).
  + **Example**: Database indexing using B-trees allows efficient search, insertion, and deletion of records, reducing time complexity to O(log N).
  + **Applications**: SQL databases (e.g., MySQL, PostgreSQL) use B-trees for indexing columns and optimizing query performance.

**4. Expression Parsing and Evaluation**

* **Arithmetic Expressions**: Trees are used to represent arithmetic expressions. The nodes in the tree represent operators and operands, and the tree structure allows easy evaluation of expressions.
  + **Example**: In an expression like 3 + (5 \* 2), you can use a **binary tree** to represent it:

markdown

CopyEdit

+

/ \

3 \*

/ \

5 2

* + **Applications**: Compilers, interpreters, and calculators use expression trees to parse and evaluate mathematical expressions.

**5. Decision Trees in Machine Learning**

* **Classification and Regression**: Decision trees are a popular machine learning algorithm used for both classification and regression tasks. They divide the dataset into subsets based on attribute values, recursively creating decision nodes until a decision is reached.
  + **Example**: A decision tree for classifying animals based on features like "has fur," "is aquatic," or "lays eggs."
  + **Applications**: Used in algorithms like **CART (Classification and Regression Trees)** and **ID3**.

**6. Routing Algorithms (e.g., in Networks)**

* **Routing Tables**: Trees are used in networking to model routing paths. For example, **routing tables** in routers are often structured as trees, where the root is the destination, and branches are the different paths that lead to that destination.
  + **Example**: In **binary trees**, the left child might represent one path, and the right child represents an alternate route. When data needs to be sent to a destination, the tree structure helps the router determine the best path.

**7. Game Trees (AI and Game Theory)**

* **Minimax Algorithm**: In AI for games (like chess or tic-tac-toe), trees are used to represent all possible moves and outcomes. The **minimax algorithm** evaluates the moves by traversing this tree and selecting the optimal move for the AI.
  + **Example**: In chess, a tree might represent the current state of the board and all possible moves. The algorithm will explore this tree to determine the best move.
  + **Applications**: Chess, Tic-Tac-Toe, Go, and other board games.

**8. Huffman Coding for Data Compression**

* **Huffman Trees**: Huffman coding is a lossless data compression algorithm that uses trees to encode data more efficiently. The tree represents the frequency of characters, and the algorithm assigns shorter codes to more frequent characters.
  + **Example**: In the string AAABBBCCCC, a Huffman tree can encode it more efficiently by assigning shorter bit sequences to more frequent characters.
  + **Applications**: File compression algorithms like **ZIP**, **JPEG**, and **MP3** use Huffman coding for efficient storage.

**9. Priority Queues (Heaps)**

* **Heap Trees**: Heaps are a type of binary tree used to implement **priority queues**. In a max-heap, the root node contains the highest value, and every parent node is greater than or equal to its children. This allows for efficient retrieval of the highest priority element.
  + **Example**: In a task scheduling system, the highest priority task is always at the root, and tasks with lower priority are at the leaves.
  + **Applications**: Implementing priority queues, scheduling algorithms, Dijkstra's algorithm for shortest paths, and event simulation.

**10. Autocomplete and Spell Check (Trie Trees)**

* **Trie Trees (Prefix Trees)**: Tries are used to store a set of strings, such that common prefixes are shared between them. This is useful in applications like autocomplete or spell-check systems, where you need to match a prefix of a word.
  + **Example**: In a search engine, when you type a few letters, the autocomplete feature uses a **trie** to find matching words quickly.
  + **Applications**: Autocomplete in search engines, dictionary lookups, spell checking, IP routing.

**11. Version Control (Merkle Trees)**

* **Merkle Trees**: Merkle trees are binary trees used in version control systems, blockchain, and cryptography. Each leaf node represents a data chunk (such as a file version), and the internal nodes represent hashes of the combined data from their children.
  + **Example**: In **blockchains**, Merkle trees are used to ensure data integrity and allow efficient verification of transactions.
  + **Applications**: Blockchains (Bitcoin, Ethereum), file versioning systems, and decentralized systems.

**12. Ternary Search Tree (TST)**

* **Optimized Searching**: Ternary search trees are used in situations where you need to store and search strings efficiently. Each node in a Ternary Search Tree has three children (left, middle, right), which provides better space optimization and faster search operations than a traditional binary tree for string-based problems.
  + **Example**: A TST can be used for **autocomplete** or **prefix-based search** in applications.
  + **Applications**: Dictionary implementations, prefix search, and word-based search engines.

**13. Network Protocols**

* **Tree Structures in Protocols**: Many network protocols, such as **DHCP**, **DNS**, or **routing protocols** like **OSPF (Open Shortest Path First)**, use tree structures to represent network topology and routes.
  + **Example**: The DNS hierarchy is represented as a tree, with root servers at the top, followed by domains, subdomains, and individual records.
  + **Applications**: Network routing, DNS resolution, IP address allocation.

**14. Real-Time Event Simulation (Discrete Event Simulation)**

* **Event Trees**: In event-driven simulation systems (like traffic simulation or real-time systems), trees are used to represent possible event sequences and their outcomes.
  + **Example**: In traffic simulation, events like "a car arrives at an intersection" or "a traffic light changes" can be represented using event trees.
  + **Applications**: Traffic simulation, game simulation, queuing models in computer network

Task 6:

 Create  a binary search operation on tree

public class BinarySearchTree {

class Node {

int value;

Node left, right;

public Node(int value) {

this.value = value;

this.left = this.right = null;

}

}

private Node root;

public BinarySearchTree() {

this.root = null;

}

// Insert method for the tree

public void insert(int value) {

root = insertRec(root, value);

}

private Node insertRec(Node root, int value) {

if (root == null) {

root = new Node(value);

return root;

}

if (value < root.value) {

root.left = insertRec(root.left, value);

} else if (value > root.value) {

root.right = insertRec(root.right, value);

}

return root;

}

// Binary Search operation

public boolean search(int value) {

return searchRec(root, value);

}

private boolean searchRec(Node root, int value) {

// Base case: root is null or value is present at the root

if (root == null) {

return false;

}

if (root.value == value) {

return true; // Value found

}

// Value is greater than root's value, so search the right subtree

if (value > root.value) {

return searchRec(root.right, value);

}

// Value is smaller than root's value, so search the left subtree

return searchRec(root.left, value);

}

// Inorder Traversal (for printing the tree)

public void inorder() {

inorderRec(root);

}

private void inorderRec(Node root) {

if (root != null) {

inorderRec(root.left);

System.out.print(root.value + " ");

inorderRec(root.right);

}

}

public static void main(String[] args) {

BinarySearchTree bst = new BinarySearchTree();

// Inserting values into the BST

bst.insert(10);

bst.insert(5);

bst.insert(15);

bst.insert(3);

bst.insert(7);

bst.insert(12);

bst.insert(20);

// Inorder Traversal

System.out.print("Inorder traversal of the BST: ");

bst.inorder(); // Should print nodes in sorted order

System.out.println();

// Perform Binary Search

int searchValue = 12;

if (bst.search(searchValue)) {

System.out.println(searchValue + " is present in the tree.");

} else {

System.out.println(searchValue + " is not present in the tree.");

}

searchValue = 25;

if (bst.search(searchValue)) {

System.out.println(searchValue + " is present in the tree.");

} else {

System.out.println(searchValue + " is not present in the tree.");

}

}

}

Ouptut: Inorder traversal of the BST: 3 5 7 10 12 15 20

12 is present in the tree.

25 is not present in the tree.