**Exploring Recursive Algorithms: Fibonacci Sequence and Factorial Calculation:**

* **Fibonacci Sequence:** In Java, you can implement a recursive function to calculate the nth Fibonacci number.

public class Fibonacci {

public static int fibonacci(int n) {

if (n <= 1) {

return n;

} else {

return fibonacci(n - 1) + fibonacci(n - 2);

}

}

public static void main(String[] args) {

int n = 5;

System.out.println("Fibonacci of " + n + " is: " + fibonacci(n)); // Output: 5

}

}

**Factorial Calculation:** Similarly, you can implement a recursive function to find the factorial of a number.

public class Factorial {

public static int factorial(int n) {

if (n == 0) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

public static void main(String[] args) {

int n = 5;

System.out.println("Factorial of " + n + " is: " + factorial(n)); // Output: 120

}

}

**Understanding Big O Notation:**

* Big O notation in Java describes the worst-case scenario for the runtime or space complexity of an algorithm.

**Example:** Calculating the Big O notation for linear search:

public class LinearSearch {

public static int linearSearch(int[] arr, int target) {

for (int i = 0; i < arr.length; i++) {

if (arr[i] == target) {

return i;

}

}

return -1;

}

public static void main(String[] args) {

int[] arr = {3, 6, 8, 2, 9, 1};

int target = 8;

System.out.println("Index of target: " + linearSearch(arr, target)); // Output: 2

}

}

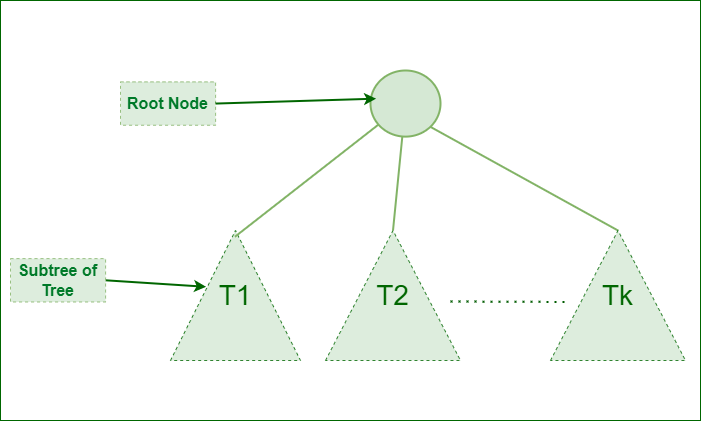


**Basic Terminologies In Tree Data Structure:**

* **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.
* **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:** A node x is a descendant of another node y if and only if y is an ancestor of y.
* **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.

**Representation of Tree Data Structure:**

*A tree consists of a root, and zero or more subtrees T1, T2, … , Tk such that there is an edge from the root of the tree to the root of each subtree.*



*Representation of Tree Data Structure*

**Representation of a Node in Tree Data Structure:**

*struct Node  
{  
   int data;  
   struct Node \*first\_child;  
   struct Node \*second\_child;  
   struct Node \*third\_child;  
   .  
   .  
   .  
   struct Node \*nth\_child;  
};*

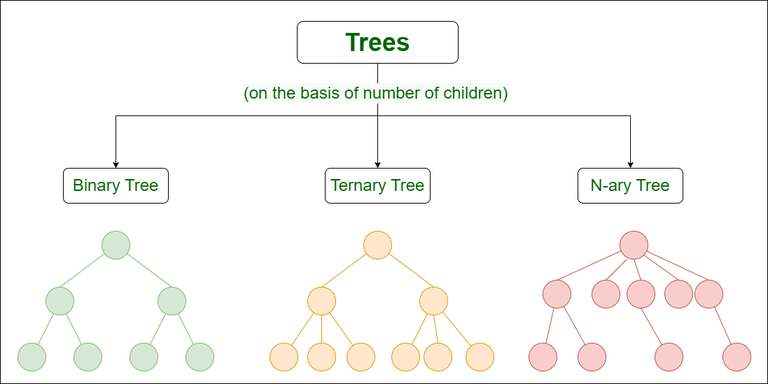
**Example of Tree data structure**



Here,

* Node 1 is the root node
* 1 is the parent of 2 and 3
* 2 and 3 are the siblings
* 4, 5, 6, and 7 are the leaf nodes
* 1 and 2 are the ancestors of 5

**Types of Tree data structures:**



* [**Binary tree**](https://www.geeksforgeeks.org/types-of-trees-in-data-structures/)**:** 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**](https://www.geeksforgeeks.org/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**](https://www.geeksforgeeks.org/generic-treesn-array-trees/)**:** 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(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

To learn more about types of trees, refer to [**this article**](https://www.geeksforgeeks.org/types-of-trees-in-data-structures/).

**Basic Operation Of Tree Data Structure:**

* **Create** – create 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**:
  + **Preorder 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.

1. **Node:**
   * A node is a fundamental building block of a tree data structure.
   * Each node in a binary tree contains a piece of data, often referred to as the "value" or "key".
   * In the provided **TreeNode** class, **val** represents the value stored in the node.
2. **Left and Right:**
   * In a binary tree, each node can have at most two children: a left child and a right child.
   * The **left** and **right** attributes of a node in the **TreeNode** class represent references to its left and right children, respectively.
   * If a node does not have a left child or right child, the corresponding attribute will be **null**.
   * The **left** and **right** attributes facilitate the hierarchical structure of the binary tree, allowing traversal and manipulation of the tree's nodes.

**Introduction to Trees, Graphs, and Hash Tables:**

* Java provides built-in data structures for trees, graphs, and hash tables in its standard library.
* **Example:** Creating a binary search tree and performing an inorder traversal:

java  
class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

}

}

public class BinaryTree {

public static void inorderTraversal(TreeNode root) {

if (root != null) {

inorderTraversal(root.left);

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

inorderTraversal(root.right);

}

}

public static void main(String[] args) {

TreeNode root = new TreeNode(5);

root.left = new TreeNode(3);

root.right = new TreeNode(7);

root.left.left = new TreeNode(2);

root.left.right = new TreeNode(4);

System.out.print("Inorder traversal: ");

inorderTraversal(root); // Output: 2 3 4 5 7

}

}

Explanation:

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int val) {

this.val = val;

}

}

1. **TreeNode Class:**
   * This class defines the structure of a tree node.
   * It has three attributes: **val** to store the value of the node, **left** to reference the left child node, and **right** to reference the right child node.
   * The constructor **TreeNode(int val)** initializes the node with the given value.

public class BinaryTree {

public static void inorderTraversal(TreeNode root) {

if (root != null) {

inorderTraversal(root.left);

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

inorderTraversal(root.right);

}

}

1. **BinaryTree Class:**
   * This class contains a static method **inorderTraversal()** for performing an inorder traversal of a binary tree.
   * Inorder traversal visits nodes in the following order: left subtree, current node, right subtree.
   * The method takes a **TreeNode** object **root** as input.
   * If the **root** node is not null, it recursively calls **inorderTraversal()** on the left child, prints the value of the current node, and then recursively calls **inorderTraversal()** on the right child.

public static void main(String[] args) {

TreeNode root = new TreeNode(5);

root.left = new TreeNode(3);

root.right = new TreeNode(7);

root.left.left = new TreeNode(2);

root.left.right = new TreeNode(4);

System.out.print("Inorder traversal: ");

inorderTraversal(root); // Output: 2 3 4 5 7

}

}

1. **Main Method:**
   * In the **main()** method, a binary tree is constructed.
   * **TreeNode** objects are created for each node, and their values are assigned accordingly.
   * The tree structure is built by setting the left and right child nodes for each node.
   * Finally, the **inorderTraversal()** method is called with the root of the tree, resulting in an inorder traversal of the binary tree.
   * The output of the inorder traversal is printed to the console.

**Advanced Sorting and Searching:**

* Java offers efficient sorting and searching algorithms in its standard library.
* **Example:** Implementing merge sort in Java

Merge sort is a popular sorting algorithm that follows the divide-and-conquer strategy to sort an array or a list. Here's an explanation of how the merge sort algorithm works:

1. **Divide:** The array to be sorted is divided into two halves recursively until each sub-array contains only one element. This is done until the array cannot be divided further, i.e., until the left index is less than the right index.
2. **Conquer:** Once the array is divided into single-element sub-arrays, the merge operation begins. In the merge operation, adjacent sub-arrays are merged together in a sorted manner to produce larger sorted sub-arrays. This process continues until all sub-arrays are merged and the entire array becomes sorted.
3. **Merge:** In the merge step, two sorted sub-arrays are merged to form a single sorted array. The merge process involves comparing elements from the two sub-arrays and placing them in the correct order into a temporary array. Once all elements are merged, the temporary array replaces the original sub-arrays, resulting in a sorted array.

Here's a high-level overview of the merge sort algorithm:

* **MergeSort(arr[], left, right):**
  + If left < right:
    - Find the middle point to divide the array into two halves:
      * mid = (left + right) / 2
    - Call MergeSort for the first half:
      * MergeSort(arr, left, mid)
    - Call MergeSort for the second half:
      * MergeSort(arr, mid + 1, right)
    - Merge the two halves sorted in step 2 and 3:
      * Merge(arr, left, mid, right)
* **Merge(arr[], left, mid, right):**
  + Create temporary arrays L[] and R[] to hold the elements of the two sub-arrays.
  + Copy data to temporary arrays L[] and R[].
  + Merge the temporary arrays back into arr[left..right]:
    - Initialize indices i, j, and k to merge the two sub-arrays.
    - Compare elements from L[] and R[] and merge them into arr[].

Merge sort has a time complexity of O(n log n), where n is the number of elements in the array. It's a stable sorting algorithm, meaning that the relative order of equal elements is preserved during sorting. Additionally, merge sort is efficient in sorting large datasets and is widely used in practice.

public class MergeSort {

public static void mergeSort(int[] arr, int left, int right) {

if (left < right) {

int mid = (left + right) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

public static void merge(int[] arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int[] L = new int[n1];

int[] R = new int[n2];

for (int i = 0; i < n1; ++i) {

L[i] = arr[left + i];

}

for (int j = 0; j < n2; ++j) {

R[j] = arr[mid + 1 + j];

}

int i = 0, j = 0;

int k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

public static void main(String[] args) {

int[] arr = {12, 11, 13, 5, 6, 7};

mergeSort(arr, 0, arr.length - 1);

System.out.println("Sorted array: " + Arrays.toString(arr)); // Output: [5, 6, 7, 11, 12, 13]

}

}

public class MergeSort {

public static void mergeSort(int[] arr, int left, int right) {

if (left < right) {

int mid = (left + right) / 2;

mergeSort(arr, left, mid);

mergeSort(arr, mid + 1, right);

merge(arr, left, mid, right);

}

}

**mergeSort Method:**

* This method implements the merge sort algorithm, which is a divide-and-conquer sorting algorithm.
* It takes an array **arr**, and two integer parameters **left** and **right**, representing the indices of the subarray to be sorted.
* The method recursively divides the array into halves until each subarray contains only one element.
* It then calls the **merge()** method to merge the sorted subarrays

public static void merge(int[] arr, int left, int mid, int right) {

int n1 = mid - left + 1;

int n2 = right - mid;

int[] L = new int[n1];

int[] R = new int[n2];

for (int i = 0; i < n1; ++i) {

L[i] = arr[left + i];

}

for (int j = 0; j < n2; ++j) {

R[j] = arr[mid + 1 + j];

}

int i = 0, j = 0;

int k = left;

while (i < n1 && j < n2) {

if (L[i] <= R[j]) {

arr[k] = L[i];

i++;

} else {

arr[k] = R[j];

j++;

}

k++;

}

while (i < n1) {

arr[k] = L[i];

i++;

k++;

}

while (j < n2) {

arr[k] = R[j];

j++;

k++;

}

}

1. **merge Method:**
   * This method merges two sorted subarrays into one sorted array.
   * It takes the original array **arr**, and the indices **left**, **mid**, and **right** indicating the boundaries of the two subarrays.
   * It calculates the lengths of the subarrays (**n1** and **n2**) and creates temporary arrays **L** and **R** to store the elements of the subarrays.
   * It then iterates through the subarrays, comparing elements and merging them into the original array **arr**.
   * Finally, it copies any remaining elements from the temporary arrays **L** and **R** into the original array **arr**.

public static void main(String[] args) {

int[] arr = {12, 11, 13, 5, 6, 7};

mergeSort(arr, 0, arr.length - 1);

System.out.println("Sorted array: " + Arrays.toString(arr)); // Output: [5, 6, 7, 11, 12, 13]

}

}

1. **main Method:**
   * In the **main()** method, an example array **arr** is initialized with unsorted elements.
   * The **mergeSort()** method is called to sort the array in ascending order.
   * Finally, the sorted array is printed to the console using **Arrays.toString()**.

**Practical Application of Algorithms:**

* Java is widely used in real-world scenarios for implementing algorithms and data structures.

**Example:** Sorting a list of strings in Java using Collections.sort():

import java.util.ArrayList;

import java.util.Collections;

public class SortingExample {

public static void main(String[] args) {

ArrayList<String> names = new ArrayList<>();

names.add("Alice");

names.add("Bob");

names.add("Charlie");

names.add("David");

names.add("Eva");

System.out.println("Before sorting: " + names);

Collections.sort(names);

System.out.println("After sorting: " + names);

}

}