|  |
| --- |
| BFS:  import java.util.\*;  class BFS {  // BFS from given source s  static ArrayList<Integer> bfs(  ArrayList<ArrayList<Integer>> adj) {  int V = adj.size();    int s = 0; // source node  // create an array to store the traversal  ArrayList<Integer> res = new ArrayList<>();    // Create a queue for BFS  Queue<Integer> q = new LinkedList<>();    // Initially mark all the vertices as not visited  boolean[] visited = new boolean[V];    // Mark source node as visited and enqueue it  visited[s] = true;  q.add(s);    // Iterate over the queue  while (!q.isEmpty()) {  int curr = q.poll();  res.add(curr);    for (int x : adj.get(curr)) {  if (!visited[x]) {  visited[x] = true;  q.add(x);  }  }  }  return res;  }    public static void main(String[] args) {    // create the adjacency list  // { {2, 3, 1}, {0}, {0, 4}, {0}, {2} }    ArrayList<ArrayList<Integer>> adj = new ArrayList<>();  adj.add(new ArrayList<>(Arrays.asList(1, 2)));  adj.add(new ArrayList<>(Arrays.asList(0, 2, 3)));  adj.add(new ArrayList<>(Arrays.asList(0, 4)));  adj.add(new ArrayList<>(Arrays.asList(1,4)));  adj.add(new ArrayList<>(Arrays.asList(2,3)));      ArrayList<Integer> ans = bfs(adj);  for (int i : ans) {  System.out.print(i + " ");  }  }  } |
| BST:  class Node {  int key;  Node left, right;  public Node(int item)  {  key = item;  left = right = null;  }  }  // BST using recursion  class BST {  // A utility function to insert a new node  // with the given key  static Node insert(Node root, int key)  {  // If the tree is empty, return a new node  if (root == null)  return new Node(key);  // If the key is already present in the tree,  // return the node  if (root.key == key)  return root;  // Otherwise, recur down the tree  if (key < root.key)  root.left = insert(root.left, key);  else  root.right = insert(root.right, key);  // Return the (unchanged) node pointer  return root;  }  // A utility function to do inorder tree traversal  static void inorder(Node root)  {  if (root != null) {  inorder(root.left);  System.out.print(root.key + " ");  inorder(root.right);  }  }  // Driver method  public static void main(String[] args)  {  Node root = null;  // Creating the following BST  // 50  // / \  // 30 70  // / \ / \  // 20 40 60 80  root = insert(root, 50);  root = insert(root, 30);  root = insert(root, 20);  root = insert(root, 40);  root = insert(root, 70);  root = insert(root, 60);  root = insert(root, 80);  // Print inorder traversal of the BST  inorder(root);  }  } |
| Bst delete iterative:  class Node {  int key;  Node left, right;  Node(int key) {  this.key = key;  this.left = this.right = null;  }  }  class BST\_del\_iter {  public static Node delIterative(Node root, int key) {  Node curr = root;  Node prev = null;  // Check if the key is actually present in the BST.  // The variable prev points to the parent of the key  // to be deleted.  while (curr != null && curr.key != key) {  prev = curr;  if (key < curr.key)  curr = curr.left;  else  curr = curr.right;  }  // Key not present  if (curr == null)  return root;  // Check if the node to be deleted has at most one child.  if (curr.left == null || curr.right == null) {  Node newCurr;  // If the left child does not exist.  if (curr.left == null)  newCurr = curr.right;  else  newCurr = curr.left;  // Check if the node to be deleted is the root.  if (prev == null)  return newCurr;  // Check if the node to be deleted is prev's left or  // right child and then replace this with newCurr.  if (curr == prev.left)  prev.left = newCurr;  else  prev.right = newCurr;  } else {    // Node to be deleted has two children.  Node p = null;  Node temp = curr.right;  while (temp.left != null) {  p = temp;  temp = temp.left;  }  if (p != null)  p.left = temp.right;  else  curr.right = temp.right;  curr.key = temp.key;  }  return root;  }  // Utility function to do inorder traversal  public static void inorder(Node root) {  if (root != null) {  inorder(root.left);  System.out.print(root.key + " ");  inorder(root.right);  }  }  // Driver code  public static void main(String[] args) {  Node root = new Node(10);  root.left = new Node(5);  root.right = new Node(15);  root.right.left = new Node(12);  root.right.right = new Node(18);  int x = 15;  root = delIterative(root, x);  inorder(root);  }  } |
| BST deletion recursive:  class Node {  int key;  Node left, right;  public Node(int item) {  key = item;  left = right = null;  }  }  class BST\_del\_rec {    // This function deletes a given key x from the  // given BST and returns the modified root of  // the BST (if it is modified).  static Node delNode(Node root, int x) {    // Base case  if (root == null) {  return root;  }  // If key to be searched is in a subtree  if (root.key > x) {  root.left = delNode(root.left, x);  } else if (root.key < x) {  root.right = delNode(root.right, x);  } else {  // If root matches with the given key  // Cases when root has 0 children or  // only right child  if (root.left == null) {  return root.right;  }  // When root has only left child  if (root.right == null) {  return root.left;  }  // When both children are present  Node succ = getSuccessor(root);  root.key = succ.key;  root.right = delNode(root.right, succ.key);  }  return root;  }    // Note that it is not a generic inorder successor  // function. It mainly works when the right child  // is not empty, which is the case we need in BST  // delete.  static Node getSuccessor(Node curr) {  curr = curr.right;  while (curr != null && curr.left != null) {  curr = curr.left;  }  return curr;  }  // Utility function to do inorder traversal  static void inorder(Node root) {  if (root != null) {  inorder(root.left);  System.out.print(root.key + " ");  inorder(root.right);  }  }  // Driver code  public static void main(String[] args) {  Node root = new Node(10);  root.left = new Node(5);  root.right = new Node(15);  root.right.left = new Node(12);  root.right.right = new Node(18);  int x = 15;  root = delNode(root, x);  inorder(root);  }  } |
| BST search iteration:  class Node {  int data;  Node left, right;  public Node(int x) {  data = x;  left = null;  right = null;  }  }  class BST\_search\_iter {    // Function to search in a bst.  static boolean search(Node root, int x) {    Node curr = root;    while (curr != null) {    // If curr node is x  if (curr.data == x)  return true;    // Search in right subtree  else if (curr.data < x)  curr = curr.right;    // Search in left subtree  else  curr = curr.left;  }    // If x is not found.  return false;  }  public static void main(String[] args) {    // Create a hard coded BST.  // 20  // / \  // 8 22  // / \  // 4 12  // / \  // 10 14  Node root = new Node(20);  root.left = new Node(8);  root.left.left = new Node(4);  root.left.right = new Node(12);  root.left.right.left = new Node(10);  root.left.right.right = new Node(14);  root.right = new Node(22);    int x = 12;  System.out.println(search(root, x));  }  } |
| BST search recursive:  class Node {  int key;  Node left, right;  public Node(int item)  {  key = item;  left = right = null;  }  }  class BST\_search\_rec {  // function to search a key in a BST  static Node search(Node root, int key)  {  // Base Cases: root is null or key is present at  // root  if (root == null || root.key == key)  return root;  // Key is greater than root's key  if (root.key < key)  return search(root.right, key);  // Key is smaller than root's key  return search(root.left, key);  }  public static void main(String[] args)  {    // Creating a hard coded tree for keeping  // the length of the code small. We need  // to make sure that BST properties are  // maintained if we try some other cases.  Node root = new Node(50);  root.left = new Node(30);  root.right = new Node(70);  root.left.left = new Node(20);  root.left.right = new Node(40);  root.right.left = new Node(60);  root.right.right = new Node(80);  // Searching for keys in the BST  System.out.println(search(root, 19) != null  ? "Found"  : "Not Found");  System.out.println(search(root, 80) != null  ? "Found"  : "Not Found");  }  }   |  | | --- | | Deletion heap:  // Java program for implement deletion in Heaps  public class deletionHeap {  // To heapify a subtree rooted with node i which is  // an index in arr[].Nn is size of heap  static void heapify(int arr[], int n, int i)  {  int largest = i; // Initialize largest as root  int l = 2 \* i + 1; // left = 2\*i + 1  int r = 2 \* i + 2; // right = 2\*i + 2  // If left child is larger than root  if (l < n && arr[l] > arr[largest])  largest = l;  // If right child is larger than largest so far  if (r < n && arr[r] > arr[largest])  largest = r;  // If largest is not root  if (largest != i) {  int swap = arr[i];  arr[i] = arr[largest];  arr[largest] = swap;  // Recursively heapify the affected sub-tree  heapify(arr, n, largest);  }  }  // Function to delete the root from Heap  static int deleteRoot(int arr[], int n)  {  // Get the last element  int lastElement = arr[n - 1];  // Replace root with first element  arr[0] = lastElement;  // Decrease size of heap by 1  n = n - 1;  // heapify the root node  heapify(arr, n, 0);  // return new size of Heap  return n;  }  /\* A utility function to print array of size N \*/  static void printArray(int arr[], int n)  {  for (int i = 0; i < n; ++i)  System.out.print(arr[i] + " ");  System.out.println();  }  // Driver Code  public static void main(String args[])  {  // Array representation of Max-Heap  // 10  // / \  // 5 3 -- 5 -> 4,3,  // / \ |  // 2 4 2  int arr[] = { 10, 5, 3, 2, 4 };  int n = arr.length;  n = deleteRoot(arr, n);  printArray(arr, n);  }  } | | Insertion heap:  // Java program for implementing insertion in Heaps  public class insertionHeap {  // Function to heapify ith node in a Heap  // of size n following a Bottom-up approach  static void heapify(int[] arr, int n, int i)  {  // Find parent  int parent = (i - 1) / 2;    if (parent >= 0) {  // For Max-Heap  // If current node is greater than its parent  // Swap both of them and call heapify again  // for the parent  if (arr[i] > arr[parent]) {    // swap arr[i] and arr[parent]  int temp = arr[i];  arr[i] = arr[parent];  arr[parent] = temp;    // Recursively heapify the parent node  heapify(arr, n, parent);  }  }  }  // Function to insert a new node to the heap.  static int insertNode(int[] arr, int n, int Key)  {  // Increase the size of Heap by 1  n = n + 1;    // Insert the element at end of Heap  arr[n - 1] = Key;    // Heapify the new node following a  // Bottom-up approach  heapify(arr, n, n - 1);    // return new size of Heap  return n;  }  /\* A utility function to print array of size n \*/  static void printArray(int[] arr, int n)  {  for (int i = 0; i < n; ++i)  System.out.println(arr[i] + " ");  System.out.println();  }  // Driver Code  public static void main(String args[])  {  // Array representation of Max-Heap  // 10  // / \  // 5 3  // / \  // 2 4    // maximum size of the array  int MAX = 1000;  int[] arr = new int[MAX];    // initializing some values  arr[0] = 10;  arr[1] = 5;  arr[2] = 3;  arr[3] = 2;  arr[4] = 4;    // Current size of the array  int n = 5;  // the element to be inserted  int Key = 15;    // The function inserts the new element to the heap and  // returns the new size of the array  n = insertNode(arr, n, Key);  printArray(arr, n);  // Final Heap will be:  // 15  // / \  // 5 10  // / \ /  // 2 4 3  }  } | | Heap operations:  import java.util.PriorityQueue;  import java.util.Collections;  public class HeapOperations {  public static void main(String[] args) {  // Min-Heap  PriorityQueue<Integer> minHeap = new PriorityQueue<>();  minHeap.add(5);  minHeap.add(1);  minHeap.add(9);  minHeap.add(3);  System.out.println("Min-Heap: " + minHeap); // Output: [1, 3, 9, 5]  System.out.println("Peek (Min): " + minHeap.peek()); // Output: 1  System.out.println("Extract (Min): " + minHeap.poll()); // Output: 1  System.out.println("Min-Heap after extract: " + minHeap); // Output: [3, 5, 9]  // Max-Heap  PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());  maxHeap.add(5);  maxHeap.add(1);  maxHeap.add(9);  maxHeap.add(3);  System.out.println("Max-Heap: " + maxHeap); // Output: [9, 5, 3, 1]  System.out.println("Peek (Max): " + maxHeap.peek()); // Output: 9  System.out.println("Extract (Max): " + maxHeap.poll()); // Output: 9  System.out.println("Max-Heap after extract: " + maxHeap); // Output: [5, 3, 1]  }  } | | BINARY search:  // Java implementation of iterative Binary Search  class BinarySearch  {  static int binarySearch(int a[], int l, int r, int x)  {  while (l <= r) {  int m = (l + r) / 2;  // Index of Element Returned  if (a[m] == x) {  return m;  // If element is smaller than mid, then  // it can only be present in left subarray  // so we decrease our r pointer to mid - 1  } else if (a[m] > x) {  r = m - 1;  // Else the element can only be present  // in right subarray  // so we increase our l pointer to mid + 1  } else {  l = m + 1;  }  }  // No Element Found  return -1;  }  public static void main(String args[])  {  int a[] = { 2, 3, 4, 10, 40 };  int n = a.length;  int x = 10;    int res = binarySearch(a, 0, n - 1, x);  System.out.println("Element to be searched is : "+ x);  if (res == -1)  System.out.println("Element is not present in array");  else  System.out.println("Element is present at index: " + res);  }  } | | LINEAR search:  // Java implementation of iterative Binary Search  class BinarySearch  {  static int binarySearch(int a[], int l, int r, int x)  {  while (l <= r) {  int m = (l + r) / 2;  // Index of Element Returned  if (a[m] == x) {  return m;  // If element is smaller than mid, then  // it can only be present in left subarray  // so we decrease our r pointer to mid - 1  } else if (a[m] > x) {  r = m - 1;  // Else the element can only be present  // in right subarray  // so we increase our l pointer to mid + 1  } else {  l = m + 1;  }  }  // No Element Found  return -1;  }  public static void main(String args[])  {  int a[] = { 2, 3, 4, 10, 40 };  int n = a.length;  int x = 10;    int res = binarySearch(a, 0, n - 1, x);  System.out.println("Element to be searched is : "+ x);  if (res == -1)  System.out.println("Element is not present in array");  else  System.out.println("Element is present at index: " + res);  }  } | | ADJCENCY matrix:  // Java Program to Implement Graph Adjacency Matrix  // Driver Class  public class AdjacencyMatrix {  // 2D array to store the adjacency matrix  private boolean[][] adjacencyMatrix;  // Number of vertices in the graph  private int numVertices;  // Constructor to initialize the graph with a given  // number of vertices  public AdjacencyMatrix(int numVertices)  {  this.numVertices = numVertices;  adjacencyMatrix  = new boolean[numVertices][numVertices];  }  // Method to add an edge between two vertices  public void addEdge(int i, int j)  {  adjacencyMatrix[i][j] = true;  // For undirected graphs  adjacencyMatrix[j][i] = true;  }  // Method to remove an edge between two vertices  public void removeEdge(int i, int j)  {  adjacencyMatrix[i][j] = false;  // For undirected graphs  adjacencyMatrix[j][i] = false;  }  // Method to check whether an edge exists between two  // vertices  public boolean hasEdge(int i, int j)  {  return adjacencyMatrix[i][j];  }  // Method to print the adjacency matrix representation  // of the graph  public void printGraph()  {  for (int i = 0; i < numVertices; i++) {  for (int j = 0; j < numVertices; j++) {  System.out.print(  adjacencyMatrix[i][j] ? "1 " : "0 ");  }  System.out.println();  }  }  // Main method to test the Graph class  public static void main(String[] args)  {  // Create a new graph with 4 vertices  AdjacencyMatrix graph = new AdjacencyMatrix(4);  // Add edges to the graph  graph.addEdge(0, 1);  graph.addEdge(1, 2);  graph.addEdge(2, 0);  graph.addEdge(1, 3);  // Print the adjacency matrix representation of the  // graph  System.out.println(  "Graph Representation (Adjacency Matrix):");  graph.printGraph();  // Check if there's an edge between vertices 0 and 1  System.out.println(  "Checking if there's an edge between vertices 0 and 1: "  + graph.hasEdge(0, 1));  // Check if there's an edge between vertices 0 and 3  System.out.println(  "Checking if there's an edge between vertices 0 and 3: "  + graph.hasEdge(0, 3));  // Remove the edge between vertices 1 and 2  graph.removeEdge(1, 2);  System.out.println(  "After removing edge between vertices 1 and 2:");  graph.printGraph();  }  } | |