|  |
| --- |
| **package** crt;  //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);  }  } |
| **package** crt;  //Java code for linearly search x in arr[]. If x  //is present then return its location, otherwise  //return -1  **class** LinearSearch  {  **static** **int** search(**int** a[], **int** n, **int** x)  {  **for** (**int** i = 0; i < n; i++) {  **if** (a[i] == x)  **return** i;  }  // return -1 if the element is not found  **return** -1;  }  **public** **static** **void** main(String[] args)  {  **int**[] a = { 3, 4, 1, 7, 5 };  **int** n = a.length;    **int** x = 4;  **int** index = *search*(a, n, x);  System.***out***.println("Element to be searched is:"+x);  **if** (index == -1)  System.***out***.println("Element is not present in the array");  **else**  System.***out***.println("Element found at index: " + index);  }  } |
| **package** crt;  //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)  {  // Insert the element at end of Heap  arr[n++] = 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  }  } |
| package crt;  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]  }  } |
| package crt;  //Java program to perform Iterative  //Depth First Traversal of Graph  import java.util.\*;  class DFS {  // Start DFS from node 0.  static ArrayList<Integer> dfs(ArrayList<ArrayList<Integer>> adj) {  int n = adj.size();    boolean[] visited = new boolean[n];  ArrayList<Integer> res = new ArrayList<>();    Stack<Integer> st = new Stack<>();  st.push(0);    while (!st.isEmpty()) {  int node = st.pop();    // If node is already visited, continue  if (visited[node] == true) {  continue;  }    // Mark this node as visited  visited[node] = true;  res.add(node);    // Traverse all edges (as stack is used, so  // push from right to left)  int size = adj.get(node).size();  for (int i = size - 1; i >= 0; i--) {  int v = adj.get(node).get(i);  if (!visited[v]) st.push(v);  }  }    return res;  }  public static void main(String[] args) {  ArrayList<ArrayList<Integer>> adj = new ArrayList<>();  adj.add(new ArrayList<>(Arrays.asList(1,2)));  adj.add(new ArrayList<>(Arrays.asList(0,2)));  adj.add(new ArrayList<>(Arrays.asList(0,1,3,4)));  adj.add(new ArrayList<>(Arrays.asList(2)));  adj.add(new ArrayList<>(Arrays.asList(2)));  ArrayList<Integer> res = dfs(adj);  for (int node : res) System.out.print(node + " ");  System.out.println();  }  } |
| **package** crt;  //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);  }  } |
| **package crt;**  **//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();**  **}**  **}** |
| **package crt;**  **//Java Program to Implement Adjacency List**  **import java.util.\*;**  **//Class representing a graph using adjacency list**  **//representation**  **class Graph {**  **private Map<Integer, List<Integer> > adjacencyList;**  **// Constructor to initialize the adjacency list**  **public Graph() {**  **adjacencyList = new HashMap<>();**  **}**  **// Method to add a new vertex to the graph**  **public void addVertex(int vertex)**  **{**  **adjacencyList.put(vertex, new ArrayList<>());**  **}**  **// Method to add an edge between two vertices**  **public void addEdge(int source, int destination)**  **{**  **adjacencyList.get(source).add(destination);**  **}**  **// Method to remove a vertex from the graph**  **public void removeVertex(int vertex)**  **{**  **adjacencyList.remove(vertex);**  **// Remove the vertex from the neighbors of other**  **// vertices**  **for (List<Integer> neighbors :**  **adjacencyList.values()) {**  **neighbors.remove(Integer.*valueOf*(vertex));**  **}**  **}**  **// Method to remove an edge between two vertices**  **public void removeEdge(int source, int destination)**  **{**  **adjacencyList.get(source).remove(**  **Integer.*valueOf*(destination));**  **// For undirected graph, uncomment below line**  **// adjacencyList.get(destination).remove(Integer.valueOf(source));**  **}**  **// Method to get the neighbors of a vertex**  **public List<Integer> getNeighbors(int vertex)**  **{**  **return adjacencyList.get(vertex);**  **}**  **// Method to print the graph**  **public void printGraph()**  **{**  **for (Map.Entry<Integer, List<Integer> > entry :**  **adjacencyList.entrySet()) {**  **System.*out*.print(entry.getKey() + " -> ");**  **for (Integer neighbor : entry.getValue()) {**  **System.*out*.print(neighbor + " ");**  **}**  **System.*out*.println();**  **}**  **}**  **}**  **public class AdjacencyList {**  **public static void main(String[] args)**  **{**  **// Create a graph**  **Graph graph = new Graph();**  **// Add vertices**  **graph.addVertex(0);**  **graph.addVertex(1);**  **graph.addVertex(2);**  **// Add edges**  **graph.addEdge(0, 1);**  **graph.addEdge(0, 2);**  **graph.addEdge(1, 2);**  **// Print the graph**  **System.*out*.println("Graph:");**  **graph.printGraph();**  **// Remove an edge and print the graph**  **graph.removeEdge(0, 1);**  **System.*out*.println("After removing edge (0, 1):");**  **graph.printGraph();**  **// Remove a vertex and print the graph**  **graph.removeVertex(2);**  **System.*out*.println("After removing vertex 2:");**  **graph.printGraph();**  **}**  **}** |