|  |
| --- |
| //AVL TREE  class Node {      int key;      Node left, right;      int height;      Node(int k) {          key = k;          left = right = null;          height = 1;      }  }  public class AVLTree {      // A utility function to get the height      // of the tree      static int height(Node N) {          if (N == null)              return 0;          return N.height;      }      // A utility function to right rotate      // subtree rooted with y      static Node rightRotate(Node y) {          Node x = y.left;          Node T2 = x.right;          // Perform rotation          x.right = y;          y.left = T2;          // Update heights          y.height = Math.max(height(y.left),                              height(y.right)) + 1;          x.height = Math.max(height(x.left),                              height(x.right)) + 1;          // Return new root          return x;      }      // A utility function to left rotate      // subtree rooted with x      static Node leftRotate(Node x) {          Node y = x.right;          Node T2 = y.left;          // Perform rotation          y.left = x;          x.right = T2;          // Update heights          x.height = Math.max(height(x.left),                              height(x.right)) + 1;          y.height = Math.max(height(y.left),                              height(y.right)) + 1;          // Return new root          return y;      }      // Get Balance factor of node N      static int getBalance(Node N) {          if (N == null)              return 0;          return height(N.left) - height(N.right);      }      static Node insert(Node node, int key) {          // 1. Perform the normal BST insertion          if (node == null)              return new Node(key);          if (key < node.key)              node.left = insert(node.left, key);          else if (key > node.key)              node.right = insert(node.right, key);          else // Duplicate keys not allowed              return node;          // 2. Update height of this ancestor node          node.height = Math.max(height(node.left),                                 height(node.right)) + 1;          // 3. Get the balance factor of this node          // to check whether this node became          // unbalanced          int balance = getBalance(node);          // If this node becomes unbalanced, then          // there are 4 cases          // Left Left Case          if (balance > 1 && key < node.left.key)              return rightRotate(node);          // Right Right Case          if (balance < -1 && key > node.right.key)              return leftRotate(node);          // Left Right Case          if (balance > 1 && key > node.left.key) {              node.left = leftRotate(node.left);              return rightRotate(node);          }          // Right Left Case          if (balance < -1 && key < node.right.key) {              node.right = rightRotate(node.right);              return leftRotate(node);          }          return node;      }      // Given a non-empty binary search tree,      // return the node with minimum key value      // found in that tree.      static Node minValueNode(Node node) {          Node current = node;          // loop down to find the leftmost leaf          while (current.left != null)              current = current.left;          return current;      }      // Recursive function to delete a node with      // given key from subtree with given root.      // It returns root of the modified subtree.      static Node deleteNode(Node root, int key) {          // STEP 1: PERFORM STANDARD BST DELETE          if (root == null)              return root;          // If the key to be deleted is smaller          // than the root's key, then it lies in          // left subtree          if (key < root.key)              root.left = deleteNode(root.left, key);          // If the key to be deleted is greater          // than the root's key, then it lies in          // right subtree          else if (key > root.key)              root.right = deleteNode(root.right, key);          // if key is same as root's key, then          // this is the node to be deleted          else {              // node with only one child or no child              if ((root.left == null) ||                  (root.right == null)) {                  Node temp = root.left != null ?                              root.left : root.right;                  // No child case                  if (temp == null) {                      temp = root;                      root = null;                  } else // One child case                      root = temp; // Copy the contents of                                   // the non-empty child              } else {                  // node with two children: Get the                  // inorder successor (smallest in                  // the right subtree)                  Node temp = minValueNode(root.right);                  // Copy the inorder successor's                  // data to this node                  root.key = temp.key;                  // Delete the inorder successor                  root.right = deleteNode(root.right, temp.key);              }          }          // If the tree had only one node then return          if (root == null)              return root;          // STEP 2: UPDATE HEIGHT OF THE CURRENT NODE          root.height = Math.max(height(root.left),                                 height(root.right)) + 1;          // STEP 3: GET THE BALANCE FACTOR OF THIS          // NODE (to check whether this node          // became unbalanced)          int balance = getBalance(root);          // If this node becomes unbalanced, then          // there are 4 cases          // Left Left Case          if (balance > 1 && getBalance(root.left) >= 0)              return rightRotate(root);          // Left Right Case          if (balance > 1 && getBalance(root.left) < 0) {              root.left = leftRotate(root.left);              return rightRotate(root);          }          // Right Right Case          if (balance < -1 && getBalance(root.right) <= 0)              return leftRotate(root);          // Right Left Case          if (balance < -1 && getBalance(root.right) > 0) {              root.right = rightRotate(root.right);              return leftRotate(root);          }          return root;      }      // A utility function to print preorder      // traversal of the tree.      static void preOrder(Node root) {          if (root != null) {              System.out.print(root.key + " ");              preOrder(root.left);              preOrder(root.right);          }      }      // Driver Code      public static void main(String[] args) {          Node root = null;          // Constructing tree given in the          // above figure          root = insert(root, 9);          root = insert(root, 5);          root = insert(root, 10);          root = insert(root, 0);          root = insert(root, 6);          root = insert(root, 11);          root = insert(root, -1);          root = insert(root, 1);          root = insert(root, 2);          System.out.println("Preorder traversal of the "                  + "constructed AVL tree is:");          preOrder(root);          root = deleteNode(root, 10);          System.out.println("\nPreorder traversal after"                  + " deletion of 10:");          preOrder(root);      }  } |
| //BUBBLE SORT  class BubbleSort {      void bubbleSort(int arr[])      {          int n = arr.length;            for (int i = 0; i < n - 1; i++)              for (int j = 0; j < n - i - 1; j++)                  if (arr[j] > arr[j + 1]) {                        // swap temp and arr[i]                      int temp = arr[j];                      arr[j] = arr[j + 1];                      arr[j + 1] = temp;                  }      }      public static void main(String args[])      {          BubbleSort ob = new BubbleSort();          int a[] = { 64, 34, 25, 12 };            ob.bubbleSort(a);            int n = a.length;            for (int i = 0; i < n; ++i)              System.out.print(a[i] + " ");          System.out.println();      }  } |
| //INSERTION SORT  public class InsertionSort {      void sort(int arr[])      {          int n = arr.length;          for (int i = 1; i < n; ++i) {              int key = arr[i];              int j = i - 1;              /\* Move elements of arr[0..i-1], that are                 greater than key, to one position ahead                 of their current position \*/              while (j >= 0 && arr[j] > key) {                  arr[j + 1] = arr[j];                  j = j - 1;              }              arr[j + 1] = key;          }      }      static void printArray(int arr[])      {          int n = arr.length;          for (int i = 0; i < n; ++i)              System.out.print(arr[i] + " ");          System.out.println();      }      public static void main(String args[])      {          int arr[] = { 12, 11, 13, 5, 6 };          InsertionSort ob = new InsertionSort();          ob.sort(arr);          printArray(arr);      }  } |
| //MERGE SORT  import java.io.\*;  class MergeSort {      // Merges two subarrays of arr[].      // First subarray is arr[l..m]      // Second subarray is arr[m+1..r]      static void merge(int arr[], int l, int m, int r)      {          // Find sizes of two subarrays to be merged          int n1 = m - l + 1;          int n2 = r - m;          // Create temp arrays          int L[] = new int[n1];          int R[] = new int[n2];          // Copy data to temp arrays          for (int i = 0; i < n1; ++i)              L[i] = arr[l + i];          for (int j = 0; j < n2; ++j)              R[j] = arr[m + 1 + j];          // Merge the temp arrays          // Initial indices of first and second subarrays          int i = 0, j = 0;          // Initial index of merged subarray array          int k = l;          while (i < n1 && j < n2) {              if (L[i] <= R[j]) {                  arr[k] = L[i];                  i++;              }              else {                  arr[k] = R[j];                  j++;              }              k++;          }          // Copy remaining elements of L[] if any          while (i < n1) {              arr[k] = L[i];              i++;              k++;          }          // Copy remaining elements of R[] if any          while (j < n2) {              arr[k] = R[j];              j++;              k++;          }      }      // Main function that sorts arr[l..r] using      // merge()      static void sort(int arr[], int l, int r)      {          if (l < r) {              // Find the middle point              int m = l + (r - l) / 2;              // Sort first and second halves              sort(arr, l, m);              sort(arr, m + 1, r);              // Merge the sorted halves              merge(arr, l, m, r);          }      }      static void printArray(int arr[])      {          int n = arr.length;          for (int i = 0; i < n; ++i)              System.out.print(arr[i] + " ");          System.out.println();      }      public static void main(String args[])      {          int arr[] = { 12, 11, 13, 5, 6, 7 };          System.out.println("Given array is");          printArray(arr);          sort(arr, 0, arr.length - 1);          System.out.println("\nSorted array is");          printArray(arr);      }  } |
| //QUICK SORT  import java.util.Arrays;  class QuickSort {      // Partition function      static int partition(int[] arr, int low, int high) {            // Choose the pivot          int pivot = arr[high];            // Index of smaller element and indicates          // the right position of pivot found so far          int i = low - 1;          // Traverse arr[low..high] and move all smaller          // elements to the left side. Elements from low to          // i are smaller after every iteration          for (int j = low; j <= high - 1; j++) {              if (arr[j] < pivot) {                  i++;                  swap(arr, i, j);              }          }            // Move pivot after smaller elements and          // return its position          swap(arr, i + 1, high);          return i + 1;      }      // Swap function      static void swap(int[] arr, int i, int j) {          int temp = arr[i];          arr[i] = arr[j];          arr[j] = temp;      }      // The QuickSort function implementation      static void quickSort(int[] arr, int low, int high) {          if (low < high) {                // pi is the partition return index of pivot              int pi = partition(arr, low, high);              // Recursion calls for smaller elements              // and greater or equals elements              quickSort(arr, low, pi - 1);              quickSort(arr, pi + 1, high);          }      }      public static void main(String[] args) {          int[] arr = {10, 7, 8, 9, 1, 5};          int n = arr.length;            quickSort(arr, 0, n - 1);            for (int val : arr) {              System.out.print(val + " ");          }      }  } |
| //ANAGRAMS  import java.util.Arrays;  class Main {    public static void main(String[] args) {      String str1 = "Race";      String str2 = "Care";        str1 = str1.toLowerCase();      str2 = str2.toLowerCase();      // check if length is same      if(str1.length() == str2.length()) {        // convert strings to char array        char[] charArray1 = str1.toCharArray();        char[] charArray2 = str2.toCharArray();        // sort the char array        Arrays.sort(charArray1);        Arrays.sort(charArray2);        // if sorted char arrays are same        // then the string is anagram        boolean result = Arrays.equals(charArray1, charArray2);        if(result) {          System.out.println(str1 + " and " + str2 + " are anagram.");        }        else {          System.out.println(str1 + " and " + str2 + " are not anagram.");        }      }      else {        System.out.println(str1 + " and " + str2 + " are not anagram.");      }    }  } |
| //SLIDING WINDOW  // // Java code O(n\*k) solution for finding maximum sum of  // // a subarray of size k  // class Main {  //     // Returns maximum sum in  //     // a subarray of size k.  //     static int maxSum(int arr[], int n, int k)  //     {  //         // Initialize result  //         int max\_sum = Integer.MIN\_VALUE;  //         // Consider all blocks starting with i.  //         for (int i = 0; i < n - k + 1; i++) {  //             int current\_sum = 0;  //             for (int j = 0; j < k; j++)  //                 current\_sum = current\_sum + arr[i + j];  //             // Update result if required.  //             max\_sum = Math.max(current\_sum, max\_sum);  //         }  //         return max\_sum;  //     }  //     // Driver code  //     public static void main(String[] args)  //     {  //         int arr[] = { 1, 4, 2, 10, 2, 3, 1, 0, 20 };  //         int k = 4;  //         int n = arr.length;  //         System.out.println(maxSum(arr, n, k));  //     }  // }  // Java code for  // O(n) solution for finding  // maximum sum of a subarray  // of size k  class SlidingWindow {      // Returns maximum sum in      // a subarray of size k.      static int maxSum(int arr[], int n, int k)      {          // n must be greater          if (n <= k) {              System.out.println("Invalid");              return -1;          }          // Compute sum of first window of size k          int max\_sum = 0;          for (int i = 0; i < k; i++)              max\_sum += arr[i];          // Compute sums of remaining windows by          // removing first element of previous          // window and adding last element of          // current window.          int window\_sum = max\_sum;          for (int i = k; i < n; i++) {              window\_sum += arr[i] - arr[i - k];              max\_sum = Math.max(max\_sum, window\_sum);          }          return max\_sum;      }      // Driver code      public static void main(String[] args)      {          int arr[] = { 1, 4, 2, 10, 2, 3, 1, 0, 20 };          int k = 4;          int n = arr.length;          System.out.println(maxSum(arr, n, k));      }  } |
| //TWO POINTER  // class Main {  //     // Function to check whether any pair exists  //     // whose sum is equal to the given target value  //     static boolean twoSum(int[] arr, int target){  //         int n = arr.length;  //         // Iterate through each element in the array  //         for (int i = 0; i < n; i++) {  //             // For each element arr[i], check every  //             // other element arr[j] that comes after it  //             for (int j = i + 1; j < n; j++) {  //                 // Check if the sum of the current pair  //                 // equals the target  //                 if (arr[i] + arr[j] == target) {  //                     return true;  //                 }  //             }  //         }  //         // If no pair is found after checking  //         // all possibilities  //         return false;  //     }  //     public static void main(String[] args){  //         int[] arr = { 0, -1, 2, -3, 1 };  //         int target = -2;  //         // Call the twoSum function and print the result  //         if (twoSum(arr, target))  //             System.out.println("true");  //         else  //             System.out.println("false");  //     }  // }  import java.util.Arrays;  class TwoPointer {      // Function to check whether any pair exists      // whose sum is equal to the given target value      static boolean twoSum(int[] arr, int target){          // Sort the array          Arrays.sort(arr);          int left = 0, right = arr.length - 1;          // Iterate while left pointer is less than right          while (left < right) {              int sum = arr[left] + arr[right];              // Check if the sum matches the target              if (sum == target)                  return true;              else if (sum < target)                  left++; // Move left pointer to the right              else                  right--; // Move right pointer to the left          }          // If no pair is found          return false;      }      public static void main(String[] args){          int[] arr = { 0, -1, 2, -3, 1 };          int target = -2;          // Call the twoSum function and print the result          if (twoSum(arr, target)) {              System.out.println("true");          }          else {              System.out.println("false");          }      }  } |
|  |