import java.util.LinkedHashMap;

import java.util.Map;

public class LRUCache {

private final int capacity;

private final LinkedHashMap<Integer, Integer> cache;

public LRUCache(int capacity) {

this.capacity = capacity;

this.cache = new LinkedHashMap<Integer, Integer>(capacity, 0.75f, true) {

protected boolean removeEldestEntry(Map.Entry<Integer, Integer> eldest) {

return size() > capacity;

}

};

}

public int get(int key) {

return cache.getOrDefault(key, -1);

}

public void put(int key, int value) {

cache.put(key, value);

}

public static void main(String[] args) {

LRUCache lruCache = new LRUCache(2);

lruCache.put(1, 1);

lruCache.put(2, 2);

System.out.println(lruCache.get(1)); //

lruCache.put(3, 3);

System.out.println(lruCache.get(2));

lruCache.put(4, 4);

System.out.println(lruCache.get(1));

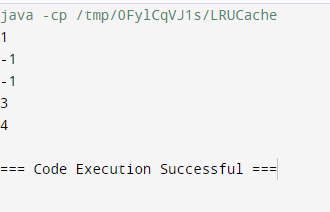
System.out.println(lruCache.get(3));

System.out.println(lruCache.get(4));

}

}

Output:



2.Program:

import java.util.ArrayList;

import java.util.ConcurrentModificationException;

import java.util.Iterator;

import java.util.List;

public class ConcurrentModificationDemo {

public static void main(String[] args) {

List<String> list = new ArrayList<>();

list.add("one");

list.add("two");

list.add("three");

list.add("four");

try {

for (String item : list) {

if (item.equals("two")) {

list.remove(item);

}

}

} catch (ConcurrentModificationException e) {

System.out.println("Caught ConcurrentModificationException: " + e);

}

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

String item = iterator.next();

if (item.equals("three")) {

iterator.remove();

}

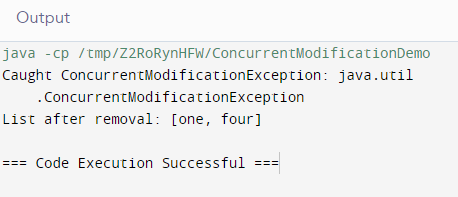
}

System.out.println("List after removal: " + list);

}

}

Output:



Program:

import java.util.Arrays;

import java.util.LinkedList;

import java.util.Queue;

public static class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) { val = x; }

}

public String serialize(TreeNode root) {

StringBuilder sb = new StringBuilder();

serializeHelper(root, sb);

return sb.toString();

}

private void serializeHelper(TreeNode node, StringBuilder sb) {

if (node == null) {

sb.append("#,");

return;

}

sb.append(node.val).append(",");

serializeHelper(node.left, sb);

serializeHelper(node.right, sb);

}

public TreeNode deserialize(String data) {

Queue<String> nodes = new LinkedList<>(Arrays.asList(data.split(",")));

return deserializeHelper(nodes);

}

private TreeNode deserializeHelper(Queue<String> nodes) {

String val = nodes.poll();

if (val.equals("#")) {

return null;

}

TreeNode node = new TreeNode(Integer.parseInt(val));

node.left = deserializeHelper(nodes);

node.right = deserializeHelper(nodes);

return node;

}

public static void main(String[] args) {

Codec codec = new Codec();

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.right = new TreeNode(3);

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

root.right.right = new TreeNode(5);

String serialized = codec.serialize(root);

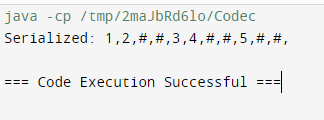
System.out.println("Serialized: " + serialized);

TreeNode deserialized = codec.deserialize(serialized);

}

}

Output:



5.

Program:

class Trie {

private TrieNode root;

private class TrieNode {

private TrieNode[] children;

private boolean isEndOfWord;

public TrieNode() {

children = new TrieNode[26]; // Each node can have 26 children (a-z)

isEndOfWord = false;

}

}

public Trie() {

root = new TrieNode();

}

public void insert(String word) {

TrieNode node = root;

for (char c : word.toCharArray()) {

int index = c - 'a';

if (node.children[index] == null) {

node.children[index] = new TrieNode();

}

node = node.children[index];

}

node.isEndOfWord = true;

}

public boolean search(String word) {

TrieNode node = searchPrefix(word);

return node != null && node.isEndOfWord;

}

public boolean startsWith(String prefix) {

return searchPrefix(prefix) != null;

}

private TrieNode searchPrefix(String prefix) {

TrieNode node = root;

for (char c : prefix.toCharArray()) {

int index = c - 'a';

if (node.children[index] == null) {

return null;

}

node = node.children[index];

}

return node;

}

public static void main(String[] args) {

Trie trie = new Trie();

trie.insert("apple");

System.out.println(trie.search("apple"));

System.out.println(trie.search("app"));

System.out.println(trie.startsWith("app"));

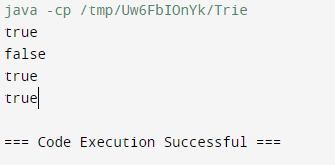
trie.insert("app");

System.out.println(trie.search("app

}

}

Output:



.

Program:

import java.util.Stack;

public class Solution {

public boolean isValid(String s) {

Stack<Character> stack = new Stack<>();

for (char c : s.toCharArray()) {

switch (c) {

case '(':

stack.push(')');

break;

case '{':

stack.push('}');

break;

case '[':

stack.push(']');

break;

case ')':

case '}':

case ']':

if (stack.isEmpty() || stack.pop() != c) {

return false;

}

break;

default:

return false;

}

}

return stack.isEmpty();

}

public static void main(String[] args) {

Solution solution = new Solution();

System.out.println(solution.isValid("()"));

System.out.println(solution.isValid("()[]{}"));

System.out.println(solution.isValid("(]"));

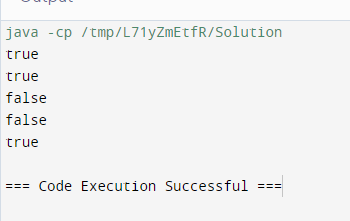
System.out.println(solution.isValid("([)]"));

System.out.println(solution.isValid("{[]}"));

}

}

Output:



7.

public class Solution {

public int maxArea(int[] height) {

int left = 0;

int right = height.length - 1;

int maxArea = 0;

while (left < right) {

int currentArea = Math.min(height[left], height[right]) \* (right - left);

maxArea = Math.max(maxArea, currentArea);

if (height[left] < height[right]) {

left++;

} else {

right--;

}

}

return maxArea;

}

public static void main(String[] args) {

Solution solution = new Solution();

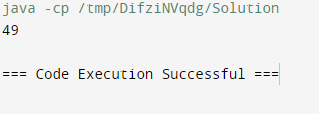
int[] height = {1, 8, 6, 2, 5, 4, 8, 3, 7};

System.out.println(solution.maxArea(height)); // Output: 49

}

}

Output:



8.

Program:

import java.util.Random;

public class Solution {

public int findKthLargest(int[] nums, int k) {

return quickSelect(nums, 0, nums.length - 1, nums.length - k);

}

private int quickSelect(int[] nums, int left, int right, int k) {

if (left == right) {

return nums[left];

}

Random random = new Random();

int pivotIndex = left + random.nextInt(right - left + 1);

pivotIndex = partition(nums, left, right, pivotIndex);

if (k == pivotIndex) {

return nums[k];

} else if (k < pivotIndex) {

return quickSelect(nums, left, pivotIndex - 1, k);

} else {

return quickSelect(nums, pivotIndex + 1, right, k);

}

}

private int partition(int[] nums, int left, int right, int pivotIndex) {

int pivotValue = nums[pivotIndex];

swap(nums, pivotIndex, right);

int storeIndex = left;

for (int i = left; i < right; i++) {

if (nums[i] < pivotValue) {

swap(nums, storeIndex, i);

storeIndex++;

}

}

swap(nums, right, storeIndex);

return storeIndex;

}

private void swap(int[] nums, int i, int j) {

int temp = nums[i];

nums[i] = nums[j];

nums[j] = temp;

}

public static void main(String[] args) {

Solution solution = new Solution();

int[] nums = {3, 2, 1, 5, 6, 4};

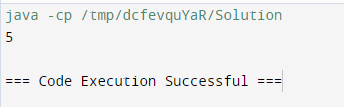
int k = 2;

System.out.println(solution.findKthLargest(nums, k)); // Output: 5

}

}

Output:



9

import java.util.ArrayList;

import java.util.List;

class IntervalTreeNode {

int start, end, max;

IntervalTreeNode left, right;

IntervalTreeNode(int start, int end) {

this.start = start;

this.end = end;

this.max = end;

this.left = null;

this.right = null;

}

}

public class IntervalTree {

private IntervalTreeNode root;

public IntervalTree() {

this.root = null;

}

public void insertInterval(int start, int end) {

root = insert(root, start, end);

}

private IntervalTreeNode insert(IntervalTreeNode node, int start, int end) {

if (node == null) {

return new IntervalTreeNode(start, end);

}

if (start < node.start) {

node.left = insert(node.left, start, end);

} else {

node.right = insert(node.right, start, end);

}

node.max = Math.max(node.max, end);

return node;

}

public void deleteInterval(int start, int end) {

root = delete(root, start, end);

}

private IntervalTreeNode delete(IntervalTreeNode node, int start, int end) {

if (node == null) {

return null;

}

if (start < node.start) {

node.left = delete(node.left, start, end);

} else if (start > node.start) {

node.right = delete(node.right, start, end);

} else if (node.end == end) {

if (node.left == null) {

return node.right;

} else if (node.right == null) {

return node.left;

}

IntervalTreeNode minNode = findMin(node.right);

node.start = minNode.start;

node.end = minNode.end;

node.right = delete(node.right, minNode.start, minNode.end);

} else {

node.right = delete(node.right, start, end);

}

node.max = Math.max(node.end, Math.max(maxValue(node.left), maxValue(node.right)));

return node;

}

private IntervalTreeNode findMin(IntervalTreeNode node) {

while (node.left != null) {

node = node.left;

}

return node;

}

private int maxValue(IntervalTreeNode node) {

return node == null ? Integer.MIN\_VALUE : node.max;

}

public List<int[]> findOverlappingIntervals(int start, int end) {

List<int[]> result = new ArrayList<>();

findOverlappingIntervals(root, start, end, result);

return result;

}

private void findOverlappingIntervals(IntervalTreeNode node, int start, int end, List<int[]> result) {

if (node == null) {

return;

}

if (doOverlap(node.start, node.end, start, end)) {

result.add(new int[]{node.start, node.end});

}

if (node.left != null && node.left.max >= start) {

findOverlappingIntervals(node.left, start, end, result);

}

findOverlappingIntervals(node.right, start, end, result);

}

private boolean doOverlap(int start1, int end1, int start2, int end2) {

return start1 <= end2 && start2 <= end1;

}

public static void main(String[] args) {

IntervalTree tree = new IntervalTree();

tree.insertInterval(15, 20);

tree.insertInterval(10, 30);

tree.insertInterval(17, 19);

tree.insertInterval(5, 20);

tree.insertInterval(12, 15);

tree.insertInterval(30, 40);

System.out.println("Overlapping intervals with [14, 16]:");

List<int[]> overlaps = tree.findOverlappingIntervals(14, 16);

for (int[] interval : overlaps) {

System.out.println("[" + interval[0] + ", " + interval[1] + "]");

}

tree.deleteInterval(10, 30);

System.out.println("Overlapping intervals with [14, 16] after deleting [10, 30]:");

overlaps = tree.findOverlappingIntervals(14, 16);

for (int[] interval : overlaps) {

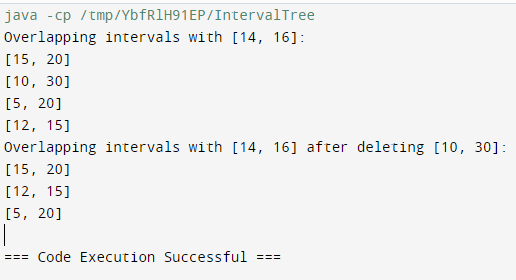
System.out.println("[" + interval[0] + ", " + interval[1] + "]");

}

}

}

Output:



10.

public class PalindromeChecker {

public static boolean isPalindrome(String s) {

String normalizedStr = s.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();

int left = 0;

int right = normalizedStr.length() - 1;

while (left < right) {

if (normalizedStr.charAt(left) != normalizedStr.charAt(right)) {

return false; // If characters don't match, it's not a palindrome

}

left++;

right--;

}

return true; }

public static void main(String[] args) {

String[] testCases = {

"A man, a plan, a canal, Panama",

"race a car",

"No lemon, no melon",

"Was it a car or a cat I saw?",

"Madam, in Eden, I'm Adam",

"Not a palindrome"

};

for (String testCase : testCases) {

System.out.println("\"" + testCase + "\" is a palindrome: " + isPalindrome(testCase));

}

}

}

Output:

