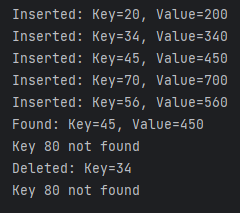
DS-LAB 11

22K-4818

Q1.

public class Task1 {  
 public static void main(String[] args) {  
 HashMap hashMap = new HashMap();  
  
 hashMap.insert(20, 200);  
 hashMap.insert(34, 340);  
 hashMap.insert(45, 450);  
 hashMap.insert(70, 700);  
 hashMap.insert(56, 560);  
  
 hashMap.search(45);  
 hashMap.search(80);  
  
 hashMap.delete(34);  
 hashMap.delete(80);  
 }  
}  
class HashNode {  
 int key;  
 int value;  
 HashNode next;  
  
 public HashNode(int key, int value) {  
 this.key = key;  
 this.value = value;  
 this.next = null;  
 }  
}  
  
class HashMap {  
 private static final int *TABLE\_SIZE* = 10;  
 private HashNode[] htable;  
  
 public HashMap() {  
 htable = new HashNode[*TABLE\_SIZE*];  
 for (int i = 0; i < *TABLE\_SIZE*; i++) {  
 htable[i] = null;  
 }  
 }  
  
 private int hashFunction(int key) {  
 return key % *TABLE\_SIZE*;  
 }  
  
 public void insert(int key, int value) {  
 int index = hashFunction(key);  
 HashNode newNode = new HashNode(key, value);  
  
 if (htable[index] == null) {  
 htable[index] = newNode;  
 } else {  
 HashNode curr = htable[index];  
 while (curr.next != null) {  
 curr = curr.next;  
 }  
 curr.next = newNode;  
 }  
  
 System.*out*.println("Inserted: Key=" + key + ", Value=" + value);  
 }  
  
 public int search(int key) {  
 int index = hashFunction(key);  
 HashNode curr = htable[index];  
  
 while (curr != null) {  
 if (curr.key == key) {  
 System.*out*.println("Found: Key=" + key + ", Value=" + curr.value);  
 return curr.value;  
 }  
 curr = curr.next;  
 }  
  
 System.*out*.println("Key " + key + " not found");  
 return -1;  
 }  
  
 public void delete(int key) {  
 int index = hashFunction(key);  
 HashNode curr = htable[index];  
 HashNode prev = null;  
  
 while (curr != null && curr.key != key) {  
 prev = curr;  
 curr = curr.next;  
 }  
  
 if (curr == null) {  
 System.*out*.println("Key " + key + " not found");  
 return;  
 }  
  
 if (prev == null) {  
 htable[index] = curr.next;  
 } else {  
 prev.next = curr.next;  
 }  
  
 System.*out*.println("Deleted: Key=" + key);  
 }  
}



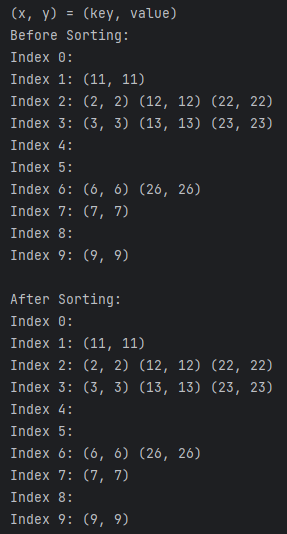
Q2.

public class Task2 {  
 public static void main(String[] args) {  
 int[] arr = {1, 5, 7, 1};  
 int K = 6;  
  
 HashMap hashMap = new HashMap();  
 int count = 0;  
 for (int num : arr) {  
 int complement = K - num;  
 int complementCount = hashMap.search(complement);  
  
 if (complementCount > 0) {  
 count += complementCount;  
 }  
  
 HashNode newNode = new HashNode(num, hashMap.search(num) + 1);  
 hashMap.insert(newNode);  
 }  
  
 System.*out*.println("Number of pairs with sum " + K + ": " + count);  
 }  
}  
class HashNode {  
 int key;  
 int frequency;  
 HashNode next;  
  
 public HashNode(int key, int frequency) {  
 this.key = key;  
 this.frequency = frequency;  
 this.next = null;  
 }  
}  
  
class HashMap {  
 private static final int *TABLE\_SIZE* = 10;  
 private HashNode[] htable;  
  
 public HashMap() {  
 htable = new HashNode[*TABLE\_SIZE*];  
 for (int i = 0; i < *TABLE\_SIZE*; i++) {  
 htable[i] = null;  
 }  
 }  
  
 private int hashFunction(int key) {  
 return key % *TABLE\_SIZE*;  
 }  
  
 public void insert(HashNode node) {  
 int index = hashFunction(node.key);  
 if (htable[index] == null) {  
 htable[index] = node;  
 } else {  
 HashNode curr = htable[index];  
 while (curr.next != null) {  
 curr = curr.next;  
 }  
 curr.next = node;  
 }  
 }  
  
 public int search(int key) {  
 int index = hashFunction(key);  
 HashNode curr = htable[index];  
  
 while (curr != null) {  
 if (curr.key == key) {  
 return curr.frequency;  
 }  
 curr = curr.next;  
 }  
  
 return 0;  
 }  
}



Q3.

public class Task3 {  
 public static void main(String[] args) {  
 int[] values = {3, 2, 9, 6, 11, 13, 7, 12, 23, 22, 26};  
 int m = 10;  
  
 HashMapWithChaining hashMap = new HashMapWithChaining();  
  
 for (int value : values) {  
 HashNode node = new HashNode(value, value);  
 hashMap.insert(node);  
 }  
  
 System.*out*.println("(x, y) = (key, value) ");  
 System.*out*.println("Before Sorting:");  
 hashMap.printHashTable();  
  
 hashMap.sortChains();  
  
 System.*out*.println("\nAfter Sorting:");  
 hashMap.printHashTable();  
 }  
}  
class HashNode {  
 int key;  
 int value;  
 HashNode next;  
  
 public HashNode(int key, int value) {  
 this.key = key;  
 this.value = value;  
 this.next = null;  
 }  
}  
  
class HashMapWithChaining {  
 private static final int *TABLE\_SIZE* = 10;  
 private HashNode[] htable;  
  
 public HashMapWithChaining() {  
 htable = new HashNode[*TABLE\_SIZE*];  
 }  
  
 private int hashFunction(int key) {  
 return key % *TABLE\_SIZE*;  
 }  
  
 public void insert(HashNode node) {  
 int index = hashFunction(node.key);  
  
 if (htable[index] == null) {  
 htable[index] = node;  
 } else {  
 HashNode curr = htable[index];  
 while (curr.next != null) {  
 curr = curr.next;  
 }  
 curr.next = node;  
 }  
 }  
  
 public void sortChains() {  
 for (int i = 0; i < *TABLE\_SIZE*; i++) {  
 if (htable[i] != null) {  
 htable[i] = sortChain(htable[i]);  
 }  
 }  
 }  
  
 private HashNode sortChain(HashNode head) {  
 HashNode sortedHead = null;  
 HashNode curr = head;  
  
 while (curr != null) {  
 HashNode next = curr.next;  
  
 if (sortedHead == null || curr.key < sortedHead.key) {  
 curr.next = sortedHead;  
 sortedHead = curr;  
 } else {  
 HashNode sortedCurrent = sortedHead;  
 while (sortedCurrent.next != null && sortedCurrent.next.key < curr.key) {  
 sortedCurrent = sortedCurrent.next;  
 }  
 curr.next = sortedCurrent.next;  
 sortedCurrent.next = curr;  
 }  
  
 curr = next;  
 }  
  
 return sortedHead;  
 }  
  
 public void printHashTable() {  
 for (int i = 0; i < *TABLE\_SIZE*; i++) {  
 HashNode curr = htable[i];  
 System.*out*.print("Index " + i + ": ");  
 while (curr != null) {  
 System.*out*.print("(" + curr.key + ", " + curr.value + ") ");  
 curr = curr.next;  
 }  
 System.*out*.println();  
 }  
 }  
}



Q4.

public class Task4 {  
 public static void main(String[] args) {  
 int[] arr = {0, -1, 2, -3, 1};  
 int result = *findTriplets*(arr);  
 System.*out*.println("arr = {0, -1, 2, -3, 1}");  
 System.*out*.println("Output: " + result);  
 }  
 private static boolean contains(int[] arr, int target, int start) {  
 for (int i = start; i < arr.length; i++) {  
 if (arr[i] == target) {  
 return true;  
 }  
 }  
 return false;  
 }  
 public static int findTriplets(int[] arr) {  
 int n = arr.length;  
  
 for (int i = 0; i < n - 1; i++) {  
 for (int j = i + 1; j < n; j++) {  
 int x = -(arr[i] + arr[j]);  
 if (*contains*(arr, x, j + 1)) {  
 return 1;  
 }  
 }  
 }  
  
 return 0;  
 }  
}



Q5.

import java.util.Arrays;  
  
public class Task5 {  
 public static void main(String[] args) {  
 char[] nuts = {'@', '%', '$', '#', '^'};  
 char[] bolts = {'%', '@', '#', '$', '^'};  
  
 *matchNutsAndBolts*(nuts, bolts);  
  
 System.*out*.println("Matched Nuts and Bolts:");  
 System.*out*.println("Nuts: " + Arrays.*toString*(nuts));  
 System.*out*.println("Bolts: " + Arrays.*toString*(bolts));  
 }  
  
 private static final char[] *ORDER* = {'!', '#', '$', '%', '&', '\*', '@', '^', '~'};  
  
 public static void matchNutsAndBolts(char[] nuts, char[] bolts) {  
 if (nuts == null || bolts == null || nuts.length != bolts.length) {  
 throw new IllegalArgumentException("Invalid input");  
 }  
  
 int[] nutIndexMap = *buildIndexMap*(nuts);  
 *quicksort*(nuts, bolts, 0, nuts.length - 1, nutIndexMap);  
 }  
  
 private static int[] buildIndexMap(char[] arr) {  
 int[] indexMap = new int[256];  
  
 for (int i = 0; i < arr.length; i++) {  
 indexMap[arr[i]] = i;  
 }  
  
 return indexMap;  
 }  
  
 private static void quicksort(char[] nuts, char[] bolts, int low, int high, int[] nutIndexMap) {  
 if (low < high) {  
 int pivotIndex = *partition*(bolts, low, high, nuts[high], nutIndexMap);  
 *partition*(nuts, low, high, bolts[pivotIndex], null);  
  
 *quicksort*(nuts, bolts, low, pivotIndex - 1, nutIndexMap);  
 *quicksort*(nuts, bolts, pivotIndex + 1, high, nutIndexMap);  
 }  
 }  
  
 private static int partition(char[] arr, int low, int high, char pivot, int[] indexMap) {  
 int i = low - 1;  
  
 for (int j = low; j < high; j++) {  
 if (*compare*(arr[j], pivot) < 0) {  
 i++;  
 *swap*(arr, i, j);  
 } else if (*compare*(arr[j], pivot) == 0) {  
 *swap*(arr, j, high);  
 j--;  
 }  
 }  
  
 i++;  
 *swap*(arr, i, high);  
  
 if (indexMap != null) {  
 int pivotIndex = indexMap[pivot];  
 *swap*(indexMap, i, pivotIndex);  
 }  
  
 return i;  
 }  
  
 private static void swap(char[] arr, int i, int j) {  
 char temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
 }  
  
 private static void swap(int[] arr, int i, int j) {  
 int temp = arr[i];  
 arr[i] = arr[j];  
 arr[j] = temp;  
 }  
  
 private static int compare(char a, char b) {  
 return *indexOf*(a) - *indexOf*(b);  
 }  
  
 private static int indexOf(char c) {  
 for (int i = 0; i < *ORDER*.length; i++) {  
 if (*ORDER*[i] == c) {  
 return i;  
 }  
 }  
 return -1;  
 }  
  
  
}

