```java
import java.util.*;

// implementing hash tables as an array of linked lists
// and using it to check whether two sequencs are permutations of each other

class Node{

    private int data;
    private Node nextNodePtr;


    public Node(){}

    public void setData(int d){
            data = d;
    }

    public  int getData(){
            return data;
    }

    public  void setNextNodePtr(Node nodePtr){
            nextNodePtr = nodePtr;
    }

    public  Node getNextNodePtr(){
        return nextNodePtr;
    }

}

class List{

    private Node headPtr;


    public  List(){
        headPtr = new Node();
        headPtr.setNextNodePtr(null);
    }


    public Node getHeadPtr(){
        return headPtr;
    }

    public  boolean isEmpty(){

        if (headPtr.getNextNodePtr() == null)
            return true;

        return false;
    }


    public  void insert(int data){

            Node currentNodePtr = headPtr.getNextNodePtr();
            Node prevNodePtr = headPtr;

            while (currentNodePtr != null){
                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr.getNextNodePtr();
            }

            Node newNodePtr = new Node();
            newNodePtr.setData(data);
            newNodePtr.setNextNodePtr(null);
            prevNodePtr.setNextNodePtr(newNodePtr);

    }

    public  void insertAtIndex(int insertIndex, int data){

            Node currentNodePtr = headPtr.getNextNodePtr();
            Node prevNodePtr = headPtr;

            int index = 0;

            while (currentNodePtr != null){

                if (index == insertIndex)
                    break;

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr.getNextNodePtr();
                index++;
            }

            Node newNodePtr = new Node();
            newNodePtr.setData(data);
            newNodePtr.setNextNodePtr(currentNodePtr);
            prevNodePtr.setNextNodePtr(newNodePtr);

    }


    public  int read(int readIndex){

            Node currentNodePtr = headPtr.getNextNodePtr();
            Node prevNodePtr = headPtr;
            int index = 0;

            while (currentNodePtr != null){

                if (index == readIndex)
                    return currentNodePtr.getData();

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr.getNextNodePtr();

                index++;

            }

        return -1; // an invalid value indicating
                    // index is out of range

    }

    public  void modifyElement(int modifyIndex, int data){

            Node currentNodePtr = headPtr.getNextNodePtr();
            Node prevNodePtr = headPtr;
            int index = 0;

            while (currentNodePtr != null){

                if (index == modifyIndex){
                    currentNodePtr.setData(data);
                    return;
                }

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr.getNextNodePtr();

                index++;
            }


    }


    public  boolean deleteElement(int data){


            Node currentNodePtr = headPtr.getNextNodePtr();
            Node prevNodePtr = headPtr;
            Node nextNodePtr = headPtr;


            while (currentNodePtr != null){

                if (currentNodePtr.getData() == data){
                    nextNodePtr = currentNodePtr.getNextNodePtr();
                    prevNodePtr.setNextNodePtr(nextNodePtr);
                    return true;
                }

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr.getNextNodePtr();

            }

            return false;

    }

    public  int countList(){

            Node currentNodePtr = headPtr.getNextNodePtr();
            int numElements = 0;

            while (currentNodePtr != null){

                numElements++;
                currentNodePtr = currentNodePtr.getNextNodePtr();

            }

            return numElements;
    }


    public  void IterativePrint(){

            Node currentNodePtr = headPtr.getNextNodePtr();

            while (currentNodePtr != null){
                System.out.print(currentNodePtr.getData()+" ");
                currentNodePtr = currentNodePtr.getNextNodePtr();
            }
```

```java

            System.out.println();

    }


    public boolean containsElement(int data){

        Node currentNodePtr = headPtr.getNextNodePtr();

        while (currentNodePtr != null){

            if (currentNodePtr.getData() == data)
                return true;

            currentNodePtr = currentNodePtr.getNextNodePtr();
        }

        return false;

    }


}


class Hashtable{

    private List[] listArray;
    private int tableSize;


    public Hashtable(int size){
        tableSize = size;
        listArray = new List[size];
        for (int index = 0; index < size; index++)
            listArray[index] = new List();
    }

    public  int getTableSize(){
        return tableSize;
    }

    public  void insert(int data){

        int hashIndex = data % tableSize;
        listArray[hashIndex].insert(data);

    }

    public void deleteElement(int data){

        int hashIndex = data % tableSize;
        listArray[hashIndex].deleteElement(data);
    }

    public  boolean hasElement(int data){

        int hashIndex = data % tableSize;
        return listArray[hashIndex].containsElement(data);

    }

    public  void printHashTable(){

        for (int hashIndex = 0; hashIndex < tableSize; hashIndex++){
                System.out.print("Hash Index: " + hashIndex + " : ");
                listArray[hashIndex].IterativePrint();
        }

    }


    public  boolean isEmpty(){

        for (int hashIndex = 0; hashIndex < tableSize; hashIndex++){

            if (!listArray[hashIndex].isEmpty())
                return false;
        }

        return true;

    }

}


class HashTableLinkedList{

    public static void main(String[] args){

    Scanner input = new Scanner(System.in);

    String integerSequence;
    System.out.print("Enter the integer sequence: ");
    integerSequence = input.nextLine();

    String testSequence;
    System.out.print("Enter the test sequence for permutation: ");
    testSequence = input.nextLine();

    int hashTableSize;
    System.out.print("Enter the size of the hash table: ");
    hashTableSize = input.nextInt();
    Hashtable hashTable = new Hashtable(hashTableSize);


    StringTokenizer stk = new StringTokenizer(integerSequence, ", ");
    while (stk.hasMoreTokens()){
        int value = Integer.parseInt(stk.nextToken());
        hashTable.insert(value);
    }

    System.out.println();

    hashTable.printHashTable();


    stk = new StringTokenizer(testSequence, ", ");
    while (stk.hasMoreTokens()){
        int testValue = Integer.parseInt(stk.nextToken());
        if (hashTable.hasElement(testValue))
            hashTable.deleteElement(testValue);
        else{
            System.out.println(testSequence + " is not a permuted sequence of " +
            integerSequence);
            return;
        }

    }

    hashTable.printHashTable();

    if (hashTable.isEmpty())
        System.out.println(testSequence +" is a permuted sequence of " + integerSequence);
    else
        System.out.println(testSequence +" is not a permuted sequence of " + integerSequence);

    }

}
```