```java
import java.util.*;

// implementing hash table as an array of linked lists
// and constructing a union of two linked lists
// the union list should have unique elements even if
// the individual linked lists may have duplicate elements

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;
```

```
65                  currentNodePtr = currentNodePtr.getNextNodePtr();
66              }
67
68              Node newNodePtr = new Node();
69              newNodePtr.setData(data);
70              newNodePtr.setNextNodePtr(null);
71              prevNodePtr.setNextNodePtr(newNodePtr);
72
73      }
74
75      public  void insertAtIndex(int insertIndex, int data){
76
77              Node currentNodePtr = headPtr.getNextNodePtr();
78              Node prevNodePtr = headPtr;
79
80              int index = 0;
81
82              while (currentNodePtr != null){
83
84                  if (index == insertIndex)
85                      break;
86
87                  prevNodePtr = currentNodePtr;
88                  currentNodePtr = currentNodePtr.getNextNodePtr();
89                  index++;
90              }
91
92              Node newNodePtr = new Node();
93              newNodePtr.setData(data);
94              newNodePtr.setNextNodePtr(currentNodePtr);
95              prevNodePtr.setNextNodePtr(newNodePtr);
96
97      }
98
99
100     public  int read(int readIndex){
101
102             Node currentNodePtr = headPtr.getNextNodePtr();
103             Node prevNodePtr = headPtr;
104             int index = 0;
105
106             while (currentNodePtr != null){
107
108                 if (index == readIndex)
109                     return currentNodePtr.getData();
110
111                 prevNodePtr = currentNodePtr;
112                 currentNodePtr = currentNodePtr.getNextNodePtr();
113
114                 index++;
115
116             }
117
118         return -1; // an invalid value indicating
119                    // index is out of range
120
121     }
122
123     public  void modifyElement(int modifyIndex, int data){
124
125             Node currentNodePtr = headPtr.getNextNodePtr();
126             Node prevNodePtr = headPtr;
127             int index = 0;
128
```

```
129             while (currentNodePtr != null){
130 
131                 if (index == modifyIndex){
132                     currentNodePtr.setData(data);
133                     return;
134                 }
135 
136                 prevNodePtr = currentNodePtr;
137                 currentNodePtr = currentNodePtr.getNextNodePtr();
138 
139                 index++;
140             }
141 
142 
143     }
144 
145 
146     public  boolean deleteElement(int data){
147 
148 
149             Node currentNodePtr = headPtr.getNextNodePtr();
150             Node prevNodePtr = headPtr;
151             Node nextNodePtr = headPtr;
152 
153 
154             while (currentNodePtr != null){
155 
156                 if (currentNodePtr.getData() == data){
157                     nextNodePtr = currentNodePtr.getNextNodePtr();
158                     prevNodePtr.setNextNodePtr(nextNodePtr);
159                     return true;
160                 }
161 
162                 prevNodePtr = currentNodePtr;
163                 currentNodePtr = currentNodePtr.getNextNodePtr();
164 
165             }
166 
167             return false;
168 
169     }
170 
171     public  int countList(){
172 
173             Node currentNodePtr = headPtr.getNextNodePtr();
174             int numElements = 0;
175 
176             while (currentNodePtr != null){
177 
178                 numElements++;
179                 currentNodePtr = currentNodePtr.getNextNodePtr();
180 
181             }
182 
183             return numElements;
184     }
185 
186 
187     public  void IterativePrint(){
188 
189             Node currentNodePtr = headPtr.getNextNodePtr();
190 
191             while (currentNodePtr != null){
192                 System.out.print(currentNodePtr.getData()+" ");
```

```
                currentNodePtr = currentNodePtr.getNextNodePtr();
            }

            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;
        while (listArray[hashIndex].deleteElement(data));

    }

    public  boolean hasElement(int data){

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

```

```
257         }
258
259         public  void printHashTable(){
260
261             for (int hashIndex = 0; hashIndex < tableSize; hashIndex++){
262                     System.out.print("Hash Index: " + hashIndex + " : ");
263                     listArray[hashIndex].IterativePrint();
264             }
265
266         }
267
268     }
269
270
271     class HashTableLinkedList{
272
273         public static void main(String[] args){
274
275         Scanner input = new Scanner(System.in);
276
277         int numElements;
278         System.out.print("Enter the number of elements you want to store in the two lists: "
            );
279         numElements = input.nextInt();
280
281         int maxValue;
282         System.out.print("Enter the maximum value for an element: ");
283         maxValue = input.nextInt();
284
285         int hashTableSize;
286         System.out.print("Enter the size of the hash table: ");
287         hashTableSize = input.nextInt();
288
289         Random randGen = new Random(System.currentTimeMillis());
290
291         List firstList = new List();
292         System.out.print("Elements generated for the first list: ");
293         for (int index = 0; index < numElements; index++){
294             int value = randGen.nextInt(maxValue);
295             firstList.insert(value);
296         }
297         firstList.IterativePrint();
298
299         List secondList = new List();
300         System.out.print("Elements generated for the second list: ");
301         for (int index = 0; index < numElements; index++){
302             int value = randGen.nextInt(maxValue);
303             secondList.insert(value);
304         }
305         secondList.IterativePrint();
306
307         // finding the union of two lists
308         // and populating the elements in a new list, unionList
309         List unionList = new List();
310         Hashtable hashTable = new Hashtable(hashTableSize);
311
312         // populating the hash table based on the first list
313         // only unique elements are inserted in the hash table
314         for (int index = 0; index < numElements; index++){
315
316             int value = firstList.read(index);
317
318             if (!hashTable.hasElement(value)){
319                 hashTable.insert(value);
```

```
320                 unionList.insert(value);
321             }
322 
323         }
324 
325         // going over the secondList and inserting only those
326         // elements that are not there in the hash table
327         // (i.e., not in the first list
328         for (int index = 0; index < numElements; index++){
329 
330             int value = secondList.read(index);
331 
332             if (!hashTable.hasElement(value)){
333                 hashTable.insert(value);
334                 unionList.insert(value);
335             }
336 
337         }
338 
339         // unionList print - only unique elements
340         System.out.print("Elements in the union list: ");
341         unionList.IterativePrint();
342 
343 
344 
345         }
346 
347     }
```

```
Enter the number of elements you want to store in the two lists: 10
Enter the maximum value for an element: 20
Enter the size of the hash table: 5
Elements generated for the first list: 4 0 7 1 0 10 19 2 1 14
Elements generated for the second list: 16 5 15 6 16 1 6 10 18 13
Elements in the union list: 4 0 7 1 10 19 2 14 16 5 15 6 18 13
```