```cpp
#include <iostream>
#include <stdlib.h> //srand, rand
#include <time.h>//clock_t, clock, CLOCKS_PER_SEC
using namespace std;

// 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;
        Node* nextNodePtr;

    public:
        Node(){}

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

        int getData(){
            return data;
        }

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

        Node* getNextNodePtr(){
            return nextNodePtr;
        }

};

class List{

    private:
        Node *headPtr;

    public:
        List(){
            headPtr = new Node();
            headPtr->setNextNodePtr(0);
        }

        Node* getHeadPtr(){
            return headPtr;
        }

        bool isEmpty(){

            if (headPtr->getNextNodePtr() == 0)
                return true;

            return false;
        }


        void insert(int data){

```

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

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

```
193             }
194 
195             return false;
196 
197         }
198 
199 
200 };
201 
202 
203 class Hashtable{
204 
205     private:
206         List* listArray;
207         int tableSize;
208 
209     public:
210         Hashtable(int size){
211             tableSize = size;
212             listArray = new List[size];
213         }
214 
215         int getTableSize(){
216             return tableSize;
217         }
218 
219         void insert(int data){
220 
221             int hashIndex = data % tableSize;
222             listArray[hashIndex].insert(data);
223 
224         }
225 
226         void deleteElement(int data){
227 
228             int hashIndex = data % tableSize;
229             while (listArray[hashIndex].deleteElement(data));
230 
231         }
232 
233         bool hasElement(int data){
234 
235             int hashIndex = data % tableSize;
236             return listArray[hashIndex].containsElement(data);
237 
238         }
239 
240         void printHashTable(){
241 
242             for (int hashIndex = 0; hashIndex < tableSize; hashIndex++){
243                 cout << "Hash Index: " << hashIndex << " : " ;
244                 listArray[hashIndex].IterativePrint();
245             }
246 
247         }
248 
249 };
250 
251 int main(){
252 
253     int numElements;
254     cout << "Enter the number of elements you want to store in the two lists: ";
255     cin >> numElements;
256 
```

```
257         int maxValue;
258         cout << "Enter the maximum value for an element: ";
259         cin >> maxValue;
260 
261         int hashTableSize;
262         cout << "Enter the size of the hash table: ";
263         cin >> hashTableSize;
264 
265         srand(time(NULL));
266 
267         List firstList;
268         cout << "Elements generated for the first list: ";
269         for (int index = 0; index < numElements; index++){
270             int value = rand() % maxValue;
271             firstList.insert(value);
272         }
273         firstList.IterativePrint();
274 
275         List secondList;
276         cout << "Elements generated for the second list: ";
277         for (int index = 0; index < numElements; index++){
278             int value = rand() % maxValue;
279             secondList.insert(value);
280         }
281         secondList.IterativePrint();
282 
283         // finding the union of two lists
284         // and populating the elements in a new list, unionList
285         List unionList;
286         Hashtable hashTable(hashTableSize);
287 
288         // populating the hash table based on the first list
289         // only unique elements are inserted in the hash table
290         for (int index = 0; index < numElements; index++){
291 
292             int value = firstList.read(index);
293 
294             if (!hashTable.hasElement(value)){
295                 hashTable.insert(value);
296                 unionList.insert(value);
297             }
298 
299         }
300 
301         // going over the secondList and inserting only those
302         // elements that are not there in the hash table
303         // (i.e., not in the first list)
304         // elements are to be included only once in the union
305         for (int index = 0; index < numElements; index++){
306 
307             int value = secondList.read(index);
308 
309             if (!hashTable.hasElement(value)){
310                 hashTable.insert(value);
311                 unionList.insert(value);
312             }
313 
314         }
315 
316         // unionList print - only unique elements
317         cout << "Elements in the union list: ";
318         unionList.IterativePrint();
319 
320 
```

```
321    return 0;
322    }
```

```
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: 7
Elements generated for the first list: 17 15 12 16 3 0 10 16 16 12
Elements generated for the second list: 8 19 17 8 13 2 1 8 18 3
Elements in the union list: 17 15 12 16 3 0 10 8 19 13 2 1 18
```