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

// implementing hash tables as an array of linked lists

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){

            Node* currentNodePtr = headPtr->getNextNodePtr();
            Node* prevNodePtr = headPtr;

            while (currentNodePtr != 0){
```

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

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

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

```
257       int hashTableSize;
258       cout << "Enter the size of the hash table: ";
259       cin >> hashTableSize;
260
261       Hashtable hashTable(hashTableSize);
262
263       srand(time(NULL));
264
265       int array[numElements];
266       cout << "Elements generated: ";
267       for (int index = 0; index < numElements; index++){
268           array[index] = rand() % maxValue;
269           cout << array[index] << " ";
270           hashTable.insert(array[index]);
271       }
272
273       cout << endl;
274
275       cout << "\nContents of the Hash Table " << endl;
276       hashTable.printHashTable();
277
278
279       int searchElement;
280       cout << "Enter an element to search: ";
281       cin >> searchElement;
282
283       if (hashTable.hasElement(searchElement))
284           cout << searchElement << " is in the original array" << endl;
285       else
286           cout << searchElement << " is not there!!" << endl;
287
288
289       int deleteElement;
290       cout << "Enter an element to delete: ";
291       cin >> deleteElement;
292       hashTable.deleteElement(deleteElement);
293
294       cout << "\nContents of the Hash Table (after delete) " << endl;
295       hashTable.printHashTable();
296
297   return 0;
```

```
Enter the number of elements you want to store in the hash table: 10
Enter the maximum value for an element: 25
Enter the size of the hash table: 7
Elements generated: 14 3 4 22 21 14 21 16 7 1

Contents of the Hash Table
Hash Index: 0 : 14 21 14 21 7
Hash Index: 1 : 22 1
Hash Index: 2 : 16
Hash Index: 3 : 3
Hash Index: 4 : 4
Hash Index: 5 :
Hash Index: 6 :
Enter an element to search: 21
21 is in the original array
Enter an element to delete: 21

Contents of the Hash Table (after delete)
Hash Index: 0 : 14 14 7
Hash Index: 1 : 22 1
Hash Index: 2 : 16
Hash Index: 3 : 3
Hash Index: 4 : 4
Hash Index: 5 :
Hash Index: 6 :
```