```
#ifndef LINKED_LIST_H
#define LINKED_LIST_H
#include <ostream>  // use cout in the print() function

using namespace std;

//----------------------------------------- nodeType

template<class Type>  //definition of node
struct nodeType
{
    Type info;
    nodeType<Type> *link;
};

//----------------------------------------- Iterator

template<class Type>     //The functions are simple, hence,
class linkedListIterator //they are included in the class definition
{
private:
    nodeType<Type> *current;

public:
    linkedListIterator()
    {  current = NULL;  }

    linkedListIterator(nodeType<Type> *ptr)
    {  current = ptr;   }

    Type operator*()
    {  return current->info;  }

    linkedListIterator<Type> operator++()  //pre-increment operator
    {
        current = current->link;
        return *this;
    }

    bool operator==(const linkedListIterator<Type>& other)
    {  return current == other.current;  }

    bool operator!=(const linkedListIterator<Type>& other)
    {  return current != other.current;  }

    //Remark: in the Java JDK, the ListIterator class has its own
    //insert and remove methods.
};

//----------------------------------------- linkedList

template<class Type>
class linkedListType
{
protected:
    int count;              // no. of elements in the list
    nodeType<Type> *first;  // pointer to the first node
```

```
58      nodeType<Type> *last;   // pointer to the last node
59
60  public:
61      //default constructor
62      linkedListType();
63
64      //copy constructor,
65      //it is used when an object is passed to a function by value
66      linkedListType(const linkedListType<Type>& other);
67
68      const linkedListType<Type>& operator=(const linkedListType<Type>& other);
69
70      ~linkedListType(); //destructor
71
72      void initializeList();
73      bool isEmptyList() const;
74      void print() const;
75      int length() const;
76      void destroyList(); //clear the list
77      virtual bool search(const Type& x) const;
78      virtual void insert(const Type& x);
79      virtual void remove(const Type& x);
80      void remove_all(const Type& x);
81
82      Type remove_front(); //remove front node and return the data value
83      //the list must be non-empty
84
85      linkedListIterator<Type> begin();
86      linkedListIterator<Type> end();
87
88      //There can be additional member functions, but to simplify
89      //the discussion, we will only consider the above functions.
90
91  private:
92      void copyList(const linkedListType<Type>& other);
93      //private function used in the copy constructor and operator=
94  };
95
96  //----------------------------------------- Implementations of member functions
97
98  template<class Type>
99  linkedListType<Type>::linkedListType()
100 {
101     count = 0;
102     first = last = NULL;
103 }
104
105
106 template<class Type>
107 void linkedListType<Type>::destroyList()
108 {  //clear the list, free the storage occupied by the nodes
109
110     nodeType<Type> *temp;
111
112     while (first != NULL)
113     {
114         temp = first;
```

```
115             first = first->link;
116             delete temp;
117         }
118         count = 0;
119         last = NULL;  //first == NULL, guaranteed by the while-loop
120     }
121
122
123     template<class Type>
124     linkedListType<Type>::~linkedListType()
125     {
126         destroyList();
127     }
128
129     template<class Type>
130     void linkedListType<Type>::copyList(const linkedListType<Type>& other)
131     {
132         if (first != NULL)
133             destroyList(); //clear the old contents of the list
134
135         if (other.first == NULL) // the other list is empty
136             return;
137
138         //copy the first node
139         first = new nodeType<Type>;
140         first->info = other.first->info;
141         first->link = NULL;
142         last = first;
143
144         //copy the remaining nodes
145         nodeType<Type> *p = other.first->link;
146
147         while (p != NULL)
148         {
149             last->link = new nodeType<Type>;
150             last = last->link;
151             last->info = p->info;
152             p = p->link;
153         }
154         last->link = NULL;
155         count = other.count;
156     }
157
158
159     template<class Type>
160     void linkedListType<Type>::initializeList()
161     {
162         count = 0;
163         first = last = NULL;
164     }
165
166
167     template<class Type>
168     linkedListType<Type>::linkedListType(const linkedListType<Type>& other)
169     {
170         initializeList();
171         copyList(other);
```

```
172  }
173
174
175  template<class Type>
176  const linkedListType<Type>& linkedListType<Type>::
177      operator=(const linkedListType<Type>& other)
178  {
179      if (this != &other) //avoid self-copy
180          copyList(other);
181
182      return *this;
183  }
184
185
186  template<class Type>
187  bool linkedListType<Type>::isEmptyList() const
188  {
189      return count == 0; // or return first == NULL;
190  }
191
192  template<class Type>
193  void linkedListType<Type>::print() const
194  {
195      nodeType<Type> *p;
196      p = first;
197      while (p != NULL)
198      {
199          cout << p->info << " ";
200          p = p->link;
201      }
202      cout << endl;
203  }
204
205
206  template<class Type>
207  int linkedListType<Type>::length() const
208  {
209      return count;
210  }
211
212
213  template<class Type>
214  bool linkedListType<Type>::search(const Type& x) const
215  {
216      nodeType<Type> *p;
217
218      p = first;
219      while (p != NULL && p->info != x)
220      {
221          p = p->link;
222      }
223      return p != NULL;
224  }
225
226
227  template<class Type>
228  void linkedListType<Type>::insert(const Type& x)
```

```
{   //append at the end of the list

    nodeType<Type> *p = new nodeType<Type>;
    p->info = x;
    p->link = NULL;

    if (first == NULL)
        first = last = p;
    else
    {
        last->link = p;
        last = p;
    }
    count++;
}


template<class Type>
void linkedListType<Type>::remove(const Type& x)
{   //remove the first instance of x in the list

    nodeType<Type> *p, *q;
    p = first;
    q = NULL;

    while (p != NULL && p->info != x)
    {
        q = p;
        p = p->link;
    }

    if (p != NULL)
    {
        if (p == first)
            first = first->link;
        else
            q->link = p->link;

        if (p == last)
            last = q;

        delete p;
        count--;
    }
}

template<class Type>
void linkedListType<Type>::remove_all(const Type& x)
{   //remove all instances of x in the list

    nodeType<Type> *p, *q;
    p = first;
    q = NULL;

    while (p != NULL)
    {
        if (p->info != x)
```

```
286           {
287               q = p;
288               p = p->link;
289           }
290           else
291           {
292               if (p == first)
293                   first = first->link;
294               else
295                   q->link = p->link;
296
297               if (p == last)
298                   last = q;
299
300               delete p;
301               count--;
302
303               if (q != NULL)
304                   p = q->link;
305               else
306                   p = first;
307           }
308       }
309   }
310
311
312   template<class Type>
313   Type linkedListType<Type>::remove_front()
314   {    //Remove the front node and return the data value.
315       //Precondition: the list must be non-empty.
316
317       nodeType<Type> *p;
318       Type x;
319
320       p = first;
321       x = first->info;  //Null-pointer exception occurs
322                         //if the list is empty
323
324       if (first == last)
325           first = last = NULL;
326       else
327           first = first->link;
328
329       delete p;
330       count--;
331
332       return x;
333   }
334
335
336   template<class Type>
337   linkedListIterator<Type> linkedListType<Type>::begin()
338   {
339       linkedListIterator<Type> temp(first);
340       //create an iterator that references the beginning of the list
341
342       return temp;
```

```
343   }
344
345
346   template<class Type>
347   linkedListIterator<Type> linkedListType<Type>::end()
348   {
349       linkedListIterator<Type> temp(NULL);
350       return temp;
351   }
352
353
354   #endif  //end of the file linkedListType.h
355
```