```
#ifndef LIST_H_INCLUDED
#define LIST_H_INCLUDED
#include <string>
#include <exception>
///Definicion
template <class T>
class List
{
public:///Ultima opcion
///class Node;
class Node
{
private:
T data;
Node* next;
public:
Node();
Node(const T&);
T& getData();
Node* getNext() const;
void setData(const T&);
void setNext(Node*);
};
class Exception: public std::exception
{
private:
std::string msg;
public:
explicit Exception(const char* message) : msg(message) { }
explicit Exception(const std::string& message) : msg(message) { }
virtual ~Exception() throw () { }
virtual const char* what() const throw () {
return msg.c_str();
}
};
private:
Node* anchor;
bool isValidPos(Node*);
void copyAll(const List&);
public:
List();
List(const List&);
~List();
bool isEmpty() const;
void insertData(Node*, const T&);
void deleteData(Node*);
Node* getFirstPos() const;
Node* getLastPos() const;
Node* getPrevPos(Node*) const;
Node* getNextPos(Node*) const;
Node* findData(const T&) const;
T& retrieve(Node*);
std::string toString() const;
void deleteAll();
List& operator = (const List&);
};
///Implementacion
using namespace std;
///Node
template <class T>
List<T>::Node::Node() : next(nullptr) { }
template <class T>
List<T>::Node::Node(const T& e) : data(e), next(nullptr) { }
template <class T>
T& List<T>::Node::getData()
{
```

```
67  return data;
68  }
69  template <class T>
70  typename List<T>::Node* List<T>::Node::getNext() const
71  {
72  return next;
73  }
74  template <class T>
75  void List<T>::Node::setData(const T& e)
76  {
77  data = e;
78  }
79  template <class T>
80  void List<T>::Node::setNext(List<T>::Node* p)
81  {
82  next = p;
83  }
84  ///List
85  template <class T>
86  bool List<T>::isValidPos(List<T>::Node* p)
87  {
88  if(isEmpty()){
89  return false;
90  }
91  Node* aux(anchor);
92  do
93  {
94  if(aux == p)
95  {
96  return true;
97  }
98  aux = aux->getNext();
99  }while(aux != anchor);
100 return false;
101 }
102 template <class T>
103 void List<T>::copyAll(const List<T>& l)
104 {
105 if(l.isEmpty()){
106 return;
107 }
108 Node* aux(l.anchor);
109 Node* lastInserted(nullptr);
110 Node* newNode;
111 do{
112 newNode = new Node(aux->getData());
113 if(newNode == nullptr)
114 {
115 throw Exception("Memoria no disponible, copyAll");
116 }
117 if(lastInserted == nullptr)
118 {
119 anchor = newNode;
120 }
121 else
122 {
123 lastInserted->setNext(newNode);
124 }
125 lastInserted = newNode;
126 aux = aux->getNext();
127 }while(aux != l.anchor);
128 lastInserted->setNext(anchor);
129 }
130 template <class T>
131 List<T>::List() : anchor(nullptr) { }
132 template <class T>
```

```
133 List<T>::List(const List<T>& l) : anchor(nullptr)
134 {
135 copyAll(l);
136 }
137 template <class T>
138 List<T>::~List()
139 {
140 deleteAll();
141 }
142 template <class T>
143 List<T>& List<T>::operator = (const List<T>& l)
144 {
145 deleteAll();
146 copyAll(l);
147 return *this;
148 }
149 template <class T>
150 bool List<T>::isEmpty() const
151 {
152 return anchor == nullptr;
153 }
154 template <class T>
155 void List<T>::insertData(List<T>::Node* p, const T& e)
156 {
157 if(p != nullptr and !isValidPos(p))
158 {
159 throw Exception(" Posicion invalida, insertData");
160 }
161 Node* aux(new Node(e));
162 if(aux == nullptr)
163 {
164 throw Exception(" Memoria no disponible, insertData");
165 }
166 if(p == nullptr)
167 {///Insertar al principio
168 if(isEmpty()){///Inserta el primer elemento
169 aux->setNext(aux);
170 }
171 else{///Hay más de un elemento
172 aux->setNext(anchor);
173 getLastPos()->setNext(aux);
174 }
175 anchor = aux;
176 }
177 else
178 {///Insertar en cualquier otra posicion
179 aux->setNext(p->getNext());
180 p->setNext(aux);
181 }
182 }
183 template <class T>
184 void List<T>::deleteData(List<T>::Node* p)
185 {
186 if(!isValidPos(p))
187 {
188 throw Exception(" Posicion invalida, deleteData.");
189 }
190 if(p == anchor)
191 {///Eliminar el primero
192 if(p->getNext() == p){///Esta solito
193 anchor = nullptr;
194 }
195 else{///Hay más de un elemento
196 getLastPos()->setNext(anchor->getNext());
197 }
198 }
```

```
199 else
200 {///Eliminar cualquiera otro
201 getPrevPos(p)->setNext(p->getNext());
202 }
203 delete p;
204 }
205 template <class T>
206 typename List<T>::Node* List<T>::getFirstPos() const
207 {
208 return anchor;
209 }
210 template <class T>
211 typename List<T>::Node* List<T>::getLastPos() const
212 {
213 if(isEmpty())
214 {
215 return nullptr;
216 }
217 Node* aux(anchor);
218 while(aux->getNext() != anchor)
219 {
220 aux = aux->getNext();
221 }
222 return aux;
223 }
224 template <class T>
225 typename List<T>::Node* List<T>::getPrevPos(List<T>::Node* p) const
226 {
227 /*if(isEmpty() or p->getNext() == anchor)
228 {
229 return nullptr;
230 }*/
231 if(isEmpty()){
232 return nullptr;
233 }
234 Node* aux(anchor);
235 do
236 {
237 if(aux->getNext() == p){
238 return aux;
239 }
240 aux = aux->getNext();
241 }while(aux != anchor);
242 return nullptr;
243 }
244 template <class T>
245 typename List<T>::Node* List<T>::getNextPos(List<T>::Node* p) const
246 {
247 /*if(!isValidPos(p) or p->getNext() == anchor)
248 {
249 return nullptr;
250 }*/
251 return p->getNext();
252 }
253 template <class T>
254 typename List<T>::Node* List<T>::findData(const T& e) const
255 {
256 if(isEmpty()){
257 return nullptr;
258 }
259 Node* aux(anchor);
260 do{
261 if(aux->getData() == e){
262 return aux;
263 }
264 aux = aux->getNext();
```

```
}while(aux != anchor);
return nullptr;
}
template <class T>
T& List<T>::retrieve(List<T>::Node* p)
{
if(!isValidPos(p))
{
throw Exception(" Posicion invalida, retireve");
}
return p->getData();
}
template <class T>
std::string List<T>::toString() const
{
std::string result;
if(!isEmpty()){
Node* aux(anchor);
do{
result += aux->getData().toString() + "\n";
aux = aux->getNext();
}while(aux != anchor);
}
return result;
}
template <class T>
void List<T>::deleteAll()
{
if(isEmpty()){
return;
}
Node* mark(anchor);
Node* aux;
do{
aux = anchor;
anchor = anchor->getNext();
delete aux;
}while(anchor != mark);
anchor = nullptr;
}

#endif // LIST_H_INCLUDED

```