```cpp
/**
 * @file CLinkedList.h
 * @author Peter Schaefer (pscha710@live.kutztown.edu)
 * @brief Class definition for a Circular Linked List, complete with iterator.
 * @version 0.1
 * @date 2023-03-11
 *
 * @copyright Copyright (c) 2023
 *
 */
#ifndef CLINKEDLIST_H
#define CLINKEDLIST_H

#include <iostream>

using namespace std;

template <typename T>
class CLinkedList;
template <typename T>
class CListItr;

/* NODE DECLARATION / DEFINITION */

/**
 * @brief A Node of type T holds data of type T and a pointer to another Node
 *
 * @tparam T
 */
template <typename T>
class Node
{
private:
  /**
   * @brief MUTATOR: Construct a new Node object
   *
   * @param info IMPORT: data held by the Node
   * @param link IMPORT: pointer to the next Node
   */
  Node(T info, Node *link = NULL) : data(info), next(link){};

  /**
   * @brief Data held by a Node
   *
   */
  T data;

  /**
   * @brief Pointer to next Node
   *
   */
  Node *next;

  friend class CLinkedList<T>;
  friend class CListItr<T>;
};

/* CLINKEDLIST DECLARATION */

/**
 * @brief A Circular Linked List, a list where the last Node points back to the first Node
 *
 * @tparam T
 */
template <typename T>
class CLinkedList
{
public:
  /**
   * @brief MUTATOR: Construct a new CLinkedList<T>::CLinkedList object
   *
   * @tparam T
   */
  CLinkedList();

```

```
76      /**
77       * @brief MUTATOR: Copies a new CLinkedList<T>::CLinkedList object
78       *
79       * @tparam T
80       * @param cl IMPORT: the list copied
81       */
82      CLinkedList(CLinkedList &cl);
83
84      /**
85       * @brief MUTATOR: Destroy the CLinkedList<T>::CLinkedList object
86       *
87       * @tparam T
88       */
89      ~CLinkedList();
90
91      /**
92       * @brief MUTATOR: Copies and assigns the right CLinkedList to this CLinkedList
93       *
94       * @tparam T
95       * @param right IMPORT: the right-side CLinkedList
96       * @return CLinkedList<T>&
97       */
98      CLinkedList &operator=(const CLinkedList &right);
99
100     /**
101      * @brief MUTATOR: Inserts data of type T into the Circular Linked List, based on its '<' value
102      *
103      * @tparam T
104      * @param data IMPORT: value of type T inserted into the Circular Linked List
105      */
106     void insert(T data);
107
108     /**
109      * @brief MUTATOR: Removes an element from the Circular Linked List
110      *
111      * @tparam T
112      * @param data IMPORT: element
113      * @return true
114      * @return false
115      */
116     bool remove(T data);
117
118     /**
119      * @brief FACILITATOR: Prints out the Circular Linked List in reverse order
120      *
121      * @tparam T
122      * @param out IMPORT:EXPORT: ostream object loaded
123      * @return ostream&
124      */
125     ostream &printReverse(ostream &out);
126
127   private:
128     /**
129      * @brief MUTATOR: Copies a Circular Linked List starting at Node n
130      *
131      * @tparam T
132      * @param n IMPORT: the head Node
133      * @return Node<T>*
134      */
135     Node<T> *copy(Node<T> *n);
136
137     /**
138      * @brief MUTATOR: Deletes the head Node and every Node following it
139      *
140      * @tparam T
141      * @param n IMPORT:EXPORT: the head Node
142      */
143     void destroy(Node<T> *, Node<T> *&);
144
145     /**
146      * @brief FACILITATOR: Called by printReverse to recursively print the Circular Linked List in reverse order
147      *
148      * @tparam T
149      * @param out IMPORT:EXPORT: ostream object loaded
150      * @param n IMPORT:EXPORT: current Node printed, recursive value
```

```
151       * @return ostream&
152       */
153      ostream &recursiveReverse(ostream &out, Node<T> *n);
154
155      /**
156       * @brief Pointer to the last Node in the Circular Linked List
157       *
158       */
159      Node<T> *last;
160
161      friend class CListItr<T>;
162    };
163
164    /**
165     * @brief FACILITATOR: Inserts the values of the Circular Linked List into a out stream object, with cascading
166     *
167     * @tparam T
168     * @param out IMPORT:EXPORT: ostream being extracted to
169     * @param right IMPORT: Circular Linked List being extracted
170     * @return ostream&
171     */
172    template <typename T>
173    ostream &operator<<(ostream &out, const CLinkedList<T> &right);
174
175    /* CLISTITR DECLARATION */
176
177    /**
178     * @brief A Circular Linked List Iterator, provides ability to iterator through the Circular Linked List
179     *
180     * @tparam T
181     */
182    template <typename T>
183    class CListItr
184    {
185    public:
186      /**
187       * @brief MUTATOR: Construct a new CListItr<T>::CListItr object
188       *
189       * @tparam T
190       * @param clist IMPORT: Circular Linked List iterator is attached to
191       */
192      CListItr(const CLinkedList<T> &clist);
193
194      /**
195       * @brief MUTATOR: Sets the iterator to the first Node of the linked list, or NULL if the list is empty
196       *
197       * @tparam T
198       */
199      void begin();
200
201      /**
202       * @brief INSPECTOR: Returns whether or not the Circular Linked List is empty
203       *
204       * @tparam T
205       * @return true
206       * @return false
207       */
208      bool isEmpty();
209
210      /**
211       * @brief INSPECTOR: Returns whether the present Node is the first Node of the Circular Linked List
212       *
213       * @tparam T
214       * @return true
215       * @return false
216       */
217      bool isFirstNode();
218
219      /**
220       * @brief INSPECTOR: Returns whether the present Node is the last Node of the circular liked list
221       *
222       * @tparam T
223       * @return true
224       * @return false
225       */
```

```
226     bool isLastNode();
227
228     /**
229      * @brief INSPECTOR:MUTATOR: Returns the data of the Node currently pointed at
230      *
231      * @tparam T
232      * @return T&
233      */
234     T &operator*();
235
236     /**
237      * @brief INSPECTOR: Returns the !const! data of the Node currently pointed at
238      *
239      * @tparam T
240      * @return const T&
241      */
242     const T &operator*() const;
243
244     /**
245      * @brief INSPECTOR:MUTATOR: Pre-increment, advances the pointer to the next Node, if there is one
246      *
247      * @return CListItr<T>&
248      */
249     CListItr<T> &operator++();
250
251     /**
252      * @brief INSPECTOR:MUTATOR: Post-increment, advances the pointer to the next Node, if there is one
253      *
254      * @return CListItr<T>
255      */
256     CListItr<T> operator++(int);
257
258   private:
259     /**
260      * @brief Constant reference to Circular Linked List that the Iterator is attached to
261      *
262      */
263     const CLinkedList<T> &clist;
264
265     /**
266      * @brief Pointer to the iterator's current Node
267      *
268      */
269     Node<T> *current;
270   };
271
272   /* CLINKEDLIST DEFINITIONS */
273
274   template <typename T>
275   CLinkedList<T>::CLinkedList() : last(NULL) {}
276
277   template <typename T>
278   CLinkedList<T>::CLinkedList(CLinkedList<T> &cl)
279   {
280     last = copy(cl.last);
281   }
282
283   template <typename T>
284   CLinkedList<T>::~CLinkedList()
285   {
286     destroy(last, last);
287   }
288
289   template <typename T>
290   CLinkedList<T> &CLinkedList<T>::operator=(const CLinkedList<T> &right)
291   {
292     // if refers to the same CLinkedList, just return the reference
293     if (this == &right)
294       return *this;
295     // they are different delete the left, and copy right into lest
296     destroy(last, last);
297     last = copy(right.last);
298     return *this;
299   }
300
```

```
template <typename T>
void CLinkedList<T>::insert(T data)
{
  if (!last) // first Node, insert data and point to itself
  {
    last = new Node<T>(data);
    last->next = last;
  }
  else if (data < last->next->data) // data is a minimum value, insert at head
  {
    last->next = new Node<T>(data, last->next);
  }
  else if (last->data < data) // data is a maximum value, insert at last
  {
    last->next = new Node<T>(data, last->next);
    last = last->next;
  }
  else // data is a middle value
  {
    Node<T> *trailP = last->next;
    Node<T> *p = last->next->next;
    for (; p != last && p->data < data; trailP = p, p = p->next)
      ;
    trailP->next = new Node<T>(data, p);
  }
}

template <typename T>
bool CLinkedList<T>::remove(T data)
{
  if (!last) // empty list, return false
    return false;
  if (last->data == data && last == last->next) // removal at last and just 1 element
  {
    delete last;
    last = NULL;
    return true;
  }
  if (last->data == data) // removal at last
  {
    Node<T> *trailLast = last;
    for (; trailLast->next != last; trailLast = trailLast->next)
      ;
    trailLast->next = last->next;
    delete last;
    last = trailLast;
    return true;
  }
  else // removal at anywhere else in the list
  {
    Node<T> *trailP = last;
    Node<T> *p = last->next;
    for (; p != last && p->data != data; trailP = p, p = p->next)
      ;
    if (p == last) // data was not found in the list, return false
      return false;
    // data was found in the list, remove and return true
    trailP->next = p->next;
    delete p;
    return true;
  }
}

template <typename T>
ostream &operator<<(ostream &out, const CLinkedList<T> &right)
{
  CListItr<T> iter(right);
  // iterator needs to refer to a linked list WITH elements
  if (!iter.isEmpty())
  {
    // extract elements until final node
    for (iter.begin(); !iter.isLastNode(); iter++)
      out << *iter << endl;
    // extract final node
    out << *iter << endl;
```

```
  }
  return out;
}

template <typename T>
ostream &CLinkedList<T>::printReverse(ostream &out)
{
  // call helper function
  // needs to refer to a linked list WITH elements
  if (last)
    recursiveReverse(out, last->next);
  return out;
}

template <typename T>
ostream &CLinkedList<T>::recursiveReverse(ostream &out, Node<T> *n)
{
  // print previous element before this element, unless last element
  if (n != last)
    recursiveReverse(out, n->next);
  // all previous elements have been printed, print this element
  out << n->data << endl;
  return out;
}

template <typename T>
Node<T> *CLinkedList<T>::copy(Node<T> *n)
{
  // if the Node is NULL, just return NULL
  if (!n)
    return NULL;

  // declare pointers to keep track of beginning for both source and destination
  Node<T> *start = n;
  // declare and copy first Node
  Node<T> *first = last = new Node<T>(n->data);
  // copy all of the remainder nodes but stop once reach the beginning again
  for (n = n->next; n != start; n = n->next, last = last->next)
    last->next = new Node<T>(n->data);
  // reassign the end of the list to the beginning Node
  last->next = first;
  return first;
}

template <class T>
void CLinkedList<T>::destroy(Node<T> *n, Node<T> *&start)
{
  if (n)
  {
    // if not back at the beginning of the list, delete the next Node
    if (n != start)
      destroy(n, start);
    // delete the passed Node and follow to the next Node
    Node<T> *doomed = n;
    n = n->next;
    delete doomed;
  }
}

/* CLISTITR DEFINITIONS */

template <typename T>
CListItr<T>::CListItr(const CLinkedList<T> &clist) : clist(clist),
                                                     current(clist.last) {}

template <typename T>
void CListItr<T>::begin()
{
  // list has at least one element
  if (clist.last)
    current = clist.last->next;
  // list has NO elements
  else
    current = NULL;
}
```

```
451
452   template <typename T>
453   bool CListItr<T>::isEmpty()
454   {
455     return clist.last == NULL;
456   }
457
458   template <typename T>
459   bool CListItr<T>::isFirstNode()
460   {
461     return current == clist.last->next;
462   }
463
464   template <typename T>
465   bool CListItr<T>::isLastNode()
466   {
467     return current == clist.last;
468   }
469
470   template <typename T>
471   T &CListItr<T>::operator*()
472   {
473     return current->data;
474   }
475
476   template <typename T>
477   const T &CListItr<T>::operator*() const
478   {
479     return current->data;
480   }
481
482   template <typename T>
483   CListItr<T> &CListItr<T>::operator++()
484   {
485     if (current)
486       current = current->next;
487     return *this;
488   }
489
490   template <typename T>
491   CListItr<T> CListItr<T>::operator++(int)
492   {
493     CListItr prior = *this;
494     ++(*this);
495     return prior;
496   }
497
498   #endif
499
```