```
#ifndef LIST_LIST_H
#define LIST_LIST_H
//
//
// Created by Oscar Maldonado on 2019-05-21.
//

#include <cassert>
#include <iostream>

using namespace std;

template <typename Obj>
class List
{
    struct Node;


    int theSize;
    Node *head;
    Node *tail;

    void init();

public:
    class const_iterator;
    class iterator;

    List() {  init();}
    List(const List & rhs);
    List(List && rhs);
    ~List();
    List & operator=(const List & rhs);
    List & operator=(List && rhs);

    iterator begin() { return head->next; }
    const_iterator begin() const { return head->next; }
    iterator end() { return tail; }
    const_iterator end() const { return tail; }

    int size() const { return theSize; }
    bool empty() const { return theSize == 0; }

    void clear();

    Obj & front() { return *begin(); }
    const Obj & front() const { return *begin(); }
    Obj & back() { return *--end(); }
    const Obj & back() const { return *--end(); }
    void push_front(const Obj & x) { insert(begin(), x); }
    void push_front(Obj && x) { insert(begin(), move(x)); }
    void push_back(const Obj & x) { insert(end(), x); }
    void push_back(Obj && x) { insert(end(), move(x)); }

    void pop_front() { erase(begin()); }
    void pop_back() { erase(--end()); }

    iterator insert(iterator, const Obj &);
    iterator insert(iterator, Obj &&);

    iterator erase(iterator);
    iterator erase(iterator, iterator);
};


template <typename Obj>
struct List<Obj>::Node
{
Obj data;
Node* next;
Node* prev;

Node():next{nullptr}, prev{nullptr}{ }
Node(const Obj& x, Node* p = nullptr, Node* n = nullptr ) : next{n},
prev{p}, data{x} { }
Node(Obj&& x, Node* p = nullptr, Node* n = nullptr) : next{n},
prev{p},data{move(x)} { }
};

template <typename Obj>
void List<Obj>::clear()
{
    while(!empty())
        List<Obj>::pop_front();
}

template <typename Obj>
class List<Obj>::const_iterator // const_iterator class
{
public:
    const_iterator(): current{nullptr} { }
    const Obj & operator*() const { return retrieve(); }

    const_iterator & operator++();
    const_iterator operator++(int);

    const_iterator & operator--();  // not in textbook
    const_iterator operator--(int); // not in textbook
    bool operator==(const const_iterator & rhs) const { return current
== rhs.current; }
    bool operator!=(const const_iterator & rhs) const { return
current != rhs.current; }

protected:
    Node *current;

    Obj& retrieve() const { return current->data; }
    const_iterator(Node* p): current{p} { }

            friend class List<Obj>;


};
template <typename Obj>
typename List<Obj>::const_iterator &
List<Obj>::const_iterator::operator++() // pre-increment
{
    current = current->next;
    return *this;
}
template<typename Obj>
typename List<Obj>::const_iterator
List<Obj>::const_iterator::operator++(int) // post-increment
{
    List<Obj>::const_iterator copy = *this;
    ++(*this);
    return copy;
}
template <typename Obj>
typename List<Obj>::const_iterator &
List<Obj>::const_iterator::operator--() // pre-decrement
{
    current = current->prev;
    return *this;
}
template <typename Obj>
typename List<Obj>::const_iterator
List<Obj>::const_iterator::operator--(int) // post-decrement
{
    List<Obj>::const_iterator copy = *this;
    --(*this);
    return copy;
}

template <typename Obj>
class List<Obj>::iterator : public List<Obj>::const_iterator //
iterator
{
```

```
public:
    iterator() { }
    Obj & operator*() { return
List<Obj>::const_iterator::retrieve(); }              // mutator
    const Obj & operator*() const { return
List<Obj>::const_iterator::operator*(); } // accessor

    iterator & operator++();
    iterator operator++(int);
 iterator & operator--();
    iterator operator--(int);

protected:
    iterator(Node *n): List<Obj>::const_iterator{n} { }
    friend class List<Obj>;


};

template <typename Obj>

typename List<Obj>::iterator  List<Obj>::iterator::operator--(int) //
post-decrement
{
    List<Obj>::iterator copy = *this;
    --(*this);
    return copy;
}
template <typename Obj>
typename List<Obj>::iterator & List<Obj>::iterator::operator--() //
pre-decrement
{
    this->current = this->current->prev;
    return *this;
}
template <typename Obj>
typename List<Obj>::iterator & List<Obj>::iterator::operator++() //
pre-decrement
{
   List<Obj>::iterator copy = *this;
   ++(*this);
   return copy;


}
template <typename Obj>
typename List<Obj>::iterator  List<Obj>::iterator::operator++(int) //
post-decrement
{
    List<Obj>::iterator copy = *this;
    --(*this);
    return copy;
}
template <typename Object>
List<Object>::List(const List & rhs)
{
    init();
    for (auto & x: rhs)
        push_back(x);
}

template <typename Object>
List<Object> & List<Object>::operator=(List && rhs)
{
    swap(theSize, rhs.theSize);
    swap(head, rhs.head);
    swap(tail, rhs.tail);

    return *this;
}

template <typename Object>
typename List<Object>::iterator
List<Object>::insert(List<Object>::iterator itr, const Object & x)
{
    Node *p = itr.current;
    Node *newNode = new Node(x, p->prev, p);
    theSize++;
    return p->prev = p->prev->next = newNode;
}
template <typename Obj>
List<Obj>::List(List&& rhs): theSize{rhs.theSize}, head{rhs.head},
tail{rhs.tail}
{
rhs.theSize = 0;
rhs.head = nullptr;
rhs.tail = nullptr;
}// List move constructor


template <typename Obj>
List<Obj>::~List()
{
    List<Obj>::clear();
    delete head;
    delete tail;
}

template <typename Obj>
typename List<Obj>::iterator List<Obj>::insert(List<Obj>::iterator
itr, Obj && x)
{
    Node* p = itr.current;
    theSize++;
    return { p->prev = p->prev->next = new Node{move(x),p->prev,p}};
}
template <typename Obj>
typename List<Obj>::iterator List<Obj>::erase(List<Obj>::iterator itr)
{
  Node* p = itr.current;
  List<Obj>::iterator retVal{p->next};
  p->prev->next = p->next;
  p->next->prev = p->prev;
  delete p;
  theSize--;
    return retVal;
}
template <typename Obj>
typename List<Obj>::iterator List<Obj>::erase(List<Obj>::iterator
from, List<Obj>::iterator to)
{
    for (List<Obj>::iterator itr = from; itr != to;)
    {
        itr = List<Obj>::erase(itr);
    }
}



template <typename Obj>
void List<Obj>::init() {
    theSize = 0;
    head = new Node();
    assert(head);

    tail = new Node();
    assert(tail);
    head->next = tail;
    tail->prev = head;
}





#endif //LIST_LIST_H
```