```
// Karthik Venkat <kv39@zips.uakron.edu>
// list.hpp: Definition of the list template and its interface.

#ifndef LIST_HPP
#define LIST_HPP
#include "test.hpp"

#include <initializer_list>
#include <iostream>

//Template struct for node
template<typename T> struct Node
{
    T value;
    Node<T> *next;
    Node<T> *prev;
    Node(T const &); //parameter of const reference to type T
};

//node constructor
template<typename T> Node<T>::Node(T const& x)
: value(x), next(nullptr), prev(nullptr)
{}

using vt = std::bidirectional_iterator_tag; //For iterators

//iterator class
template<typename valtype, typename T> struct const_iterator
: std::iterator<vt, T>
{

    Node<T> *update; //updated postition of iter

    T& getme() const
    {
      return update->value; //Returns value of node
    }

    const_iterator(Node<T>* s) //constructor for const iterator
    :update(s)
    {}

    const_iterator()
    :update(nullptr)
    {}

    T const& operator *() const //Overload for dereferencing operator
    {
      return getme();
    }

    bool operator != (const const_iterator& a) const //inequality overload
    {
      if (*this == a) return false;
      return true;
    }

    bool operator == (const const_iterator& a) const //Equality overload
    {
      if (update == a.update) return true;
      return false;
    }
```



```
    const_iterator& operator--() //prefix decrement overload
    {
       update = update->prev;
       return *this;
    }

    const_iterator operator--(int)   //postfix decremet overload
    {
      const_iterator temp = *this;
        --(*this);
        return temp;
    }

    const_iterator& operator++() //Prefix increment overload
    {
       update = update->next;
       return *this;
    }

    const_iterator operator++(int) //postfix increment overload
    {
      const_iterator temp = *this;
        ++(*this);
        return temp;
    }
};

template<typename T> struct iterator : public const_iterator<vt, T>
{
    iterator(Node<T>* s) //constructor for iterator
    :const_iterator<vt, T>(s)
    {}

    T& operator*() //dereferencing overload
    {
      return const_iterator<vt,T>::getme();
    }

    const T& operator*() const //dereferencing overload
    {
       return const_iterator<vt,T>::operator*();
    }

    iterator& operator ++ () //prefix increment overload
    {
      this->update = this->update->next;
      return *this;
     }


    iterator operator ++ (int) //Postfix increment overload
    {
      iterator temp = *this;
      ++(*this);
      return temp;
    }

    iterator& operator -- () //prefix decrement overload
    {
       this->update = this->update->prev;
        return *this;
```

```
    }

    iterator operator -- (int) //postfix decrement overload
    {
      iterator temp = *this;
        --(*this);
        return temp;
    }
};


template<typename T> struct List //class for list
{
    using constant_iter = const_iterator<vt, T>;
    using iter = iterator<T>;

    Node<T> *head;
    Node<T> *tail;

    public:
    iter begin()
    {
       return iter(head->next);
    }

    constant_iter begin() const
    {
      return constant_iter(head->next);
    }

    iter end()
    {
       return iter(tail);
    }

    constant_iter end() const
    {
      return constant_iter(tail);
    }

    std::size_t nodes = 0; //size store
    List(); //default constructor
    List(std::initializer_list<T>); //initializer list constructor
    List(List<T> const&); //copy constructor
    ~List(); //Destructor

    std::size_t size() const; //returns size
    void clear(); //clears contents of list
    void push_back(T const&); //push element to back of list
    void push_front(T const&); //push to front of list
    void pop_front(); //Pops front element
    void pop_back(); //pops last element

    bool empty() const; //returns true if empty and false otherwise

    T& front(); //returns address to front element
    T& front() const; //returns const reference to front element
    T& back(); //returns address to back element
    T& back() const; //returns const reference to back element

    List<T>& operator = (List<T> const&); //copy assign
};
```



```
template<typename T>//Split for neatness
int compare (Node<T> *first1, Node<T> *limit1, Node<T> *first2, Node<T> *lim
;

template<typename T> List<T>::List() //Default constructor for list
:head(nullptr), tail(nullptr)
{}

template<typename T> List<T>::List(std::initializer_list<T> list)
:head(nullptr), tail(nullptr) //necessary initialization of members.
{
    for (T const& elem : list) push_back(elem);
}

template<typename T> List<T>::List(List<T> const &l)  //copy constructor
:head(nullptr) ,tail(nullptr)
{
    Node<T> *m = l.head, *follow;
    if(!l.head) head = m;
    else
    {
       bool first = true; //first
        while(m) //While m is not null
        {
            Node<T> *n = new Node<T>(m->value); //new node which moves to m
            n->prev = follow; //follow n to prev
            if(first) //if first is not null
            {
              head = n; //head is assigned n
              first = false; //not first
            }
            follow->next = n; //follow to next
            follow = n;
            m = m->next; //m moves to next
        }
        tail = follow; //set tail to follow
    }
}

template<typename T> List<T>::~List() //destructor
{
  clear(); //empties list
}

template<typename T> List<T>& List<T>::operator = (List<T> const &L)
{
  if(this == &L)
  return *this;
  Node<T> *iterator = L.head;
  while (iterator)
 {
   T obj = iterator->value;
   push_back(obj);
   iterator = iterator->next;
 }
  return *this;
}

template<typename T> void List<T>::push_front(T const& x)
{
    Node<T> *m = new Node<T>(x);
    if (!head) //if head is null
```

```
    {
        ++nodes; //increment size of list
        tail = m;
        head = m;
    }

    else
    {
        ++nodes; //increment size of list
        m->next = head; //next node after m takes value of head
        head->prev = m; //previous node from head now takes m
        head = m; //new head is assigned m
    }
}

template<typename T> void List<T>::push_back(T const& x)
{
    Node<T> *m = new Node<T>(x);
    if (!tail) //if tail is null
    {
      ++nodes; //increment size of list by 1
      head = m;
      tail = m;
    }
    else
    {
        ++nodes; //increment size of list by 1
        m->prev = tail; //previous node before m takes value of tail
        tail->next = m; //next node from tail takes value of m
        tail = m; //new head is assigned m
    }
}

template<typename T> void List<T>::pop_back()
{
    assert(!empty()); //check for empty list
    Node<T> *m = tail; //m points to tail
    m = tail->prev; // set m
    delete tail;
    m->next = nullptr; //null terminator for list
    tail = m; //update tail
    --nodes; //Decrement size of list by 1
}

template<typename T> void List<T>::pop_front()
{
    assert(!empty()); //check for empty list
    Node<T> *m = head; //m points to head
    m = head->next; //set m
    delete head;
    m->prev = nullptr; //null terminate for list
    head = m; //update head
    --nodes; //decrement size of list by 1
}

//front and back functions
template<typename T> T& List<T>::front()
{
  assert(!empty()); //checks for empty list
  return head->value;
}
```



```
template<typename T> T& List<T>::front() const
{
  assert(!empty()); //checks for empty list
  return head->value;
}

template<typename T> T& List<T>::back()
{
  assert(!empty()); //checks for empty list
  return tail-> value;
}

template<typename T> T& List<T>::back() const
{
   assert(!empty()); //checks for empty list
   return tail-> value;
 }


template<typename T> bool List<T>::empty() const
{
  if (size() == 0) return true; //return true if size is 0
  return false;
}

template<typename T> std::size_t List<T>::size() const
{
    size_t count = 0; //initialize size to be 0
    Node<T> *m = head;
    while(m) //While p is not null
     {
        ++count; //increment count
        m = m->next; //move pointer to next node
      }
    return count; //return size of list
}

template<typename T> void List<T>::clear()
{
    Node<T> *m = head;
    while(m)
    {
        Node<T> *n = m->next; //move to next element
        delete m; // delete old element
        m = n; //update m
        nodes = 0; //update size of list to 0
    }
    head = nullptr; //reset head
    tail = nullptr; //reset tail
}

template<typename T>//Split for neatness
int compare (Node<T> *first1, Node<T> *limit1, Node<T> *first2, Node<T> *lim
{
  while (first1 != limit1 && first2 != limit2)
  {
    if (first1->value < first2->value) return -1; //if a<b
    if (first1->value > first2->value) return 1; // if a>b
    first1 = first1->next; //increment first list
    first2 = first2->next; //increment second list
  }
  if (first1 == limit1)
```

```
  {
    if (first2 != limit2) return -1; //if a<b
    else return 0; //if a==b
  }
    else return 1; // if a>b
}

//Overload for equality
template<typename T> bool operator == (List<T> const &L1, List<T> const &L2)
{
  if (compare(L1.head,L1.tail, L2.head, L2.tail) == 0) return true;
  return false;
}

//overload for inequality
template<typename T> bool operator != (List<T> const& a, List<T> const& b)
{
   if (a == b) return false;
   return true;
}

//Comparing operators implemented using iterators
//overload for less than
template<typename T> bool operator < (List<T> const& a, List<T> const& b)
{
  return std::lexicographical_compare(a.begin(), a.end(), b.begin(), b.end());
}

//Overload for greater than
template<typename T> bool operator > (List<T> const& a, List<T> const& b)
{
   if ((a < b) || (a == b)) return false;
   return true;
}

//overload for less than or equal to
template<typename T> bool operator <= (List<T> const& a, List<T> const& b)
{
   if ((a < b) || (a == b)) return true;
   return false;
}

//overload for greate than or equal to
template<typename T> bool operator >= (List<T> const& a, List<T> const& b)
{
   if ((a > b) || (a == b)) return true;
   return false;
}

#endif
```



```
// Karthik Venkat <kv39@zips.uakron.edu>
//
// list.hpp: This file is intentionally empty.

#include "list.hpp"
```