```cpp
#pragma once

#include "DoublyLinkedList.h"
#include "DoublyLinkedListIterator.h"

template<typename T>
class List
{
private:
    using Node = typename DoublyLinkedList<T>::Node;

    Node fHead; // first element
    Node fTail; // last element
    size_t fSize; // number of elements

public:
    using Iterator = DoublyLinkedListIterator<T>;

    List() noexcept : fHead(nullptr), fTail(nullptr), fSize(0) {} //
     default constructor (2)

    // Copy semantics
    List(const List& aOther) // copy constructor (10)
        : fHead(nullptr), fTail(nullptr), fSize(0)
    {
        for (Node current = aOther.fHead; current; current =
         current->fNext)
        {
            push_back(current->fData);
        }
    }

    List& operator=(const List& aOther) // copy assignment (14)
    {
        if (this != &aOther)
        {
            List temp(aOther);
            swap(temp);
        }
        return *this;
    }

    // Move semantics
    List(List&& aOther) noexcept // move constructor (4)
        : fHead(std::move(aOther.fHead)),
         fTail(std::move(aOther.fTail)), fSize(aOther.fSize)
    {
        aOther.fHead = nullptr;
        aOther.fTail = nullptr;
        aOther.fSize = 0;
    }
```

```
List& operator=(List&& aOther) noexcept // move assignment (8)
{
    if (this != &aOther)
    {
        swap(aOther);
        aOther.fHead = nullptr;
        aOther.fTail = nullptr;
        aOther.fSize = 0;
    }
    return *this;
}

void swap(List& aOther) noexcept // swap elements (9)
{
    std::swap(fHead, aOther.fHead);
    std::swap(fTail, aOther.fTail);
    std::swap(fSize, aOther.fSize);
}

// Basic operations
size_t size() const noexcept { return fSize; } // list size (2)

template<typename U>
void push_front(U&& aData) // add element at front (24)
{
    Node newNode =
     DoublyLinkedList<T>::makeNode(std::forward<U>(aData));
    newNode->fNext = fHead;
    if (fHead)
    {
        fHead->fPrevious = newNode;
    }
    fHead = newNode;
    if (!fTail)
    {
        fTail = fHead;
    }
    ++fSize;
}

template<typename U>
void push_back(U&& aData) // add element at back (24)
{
    Node newNode =
     DoublyLinkedList<T>::makeNode(std::forward<U>(aData));
    newNode->fPrevious = fTail;
    if (fTail)
    {
        fTail->fNext = newNode;
    }
```

```
    fTail = newNode;
    if (!fHead)
    {
        fHead = fTail;
    }
    ++fSize;
}

void remove(const T& aElement) noexcept // remove element (36)
{
    Node current = fHead;
    while (current)
    {
        if (current->fData == aElement)
        {
            if (current == fHead)
            {
                fHead = current->fNext;
            }
            if (current == fTail)
            {
                fTail = current->fPrevious.lock();
            }
            current->isolate();
            --fSize;
            return;
        }
        current = current->fNext;
    }
}

const T& operator[](size_t aIndex) const // list indexer (14)
{
    Node current = fHead;
    for (size_t i = 0; i < aIndex; ++i)
    {
        current = current->fNext;
    }
    return current->fData;
}

// Iterator interface
Iterator begin() const noexcept // (4)
{
    return Iterator(fHead, fTail).begin();
}

Iterator end() const noexcept // (4)
{
    return Iterator(fHead, fTail).end();
}
```

```
    Iterator rbegin() const noexcept // (4)
    {
        return Iterator(fHead, fTail).rbegin();
    }

    Iterator rend() const noexcept // (4)
    {
        return Iterator(fHead, fTail).rend();
    }
};
```