Task:

Using the template classes DoublyLinkedList and DoublyLinkedListIterator, implement the template class List as specified below:  
#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; // default constructor (2)  
// copy semantics  
List( const List& aOther ); // copy constructor (10)  
List& operator=( const List& aOther ); // copy assignment (14)  
// move semantics  
List( List&& aOther ) noexcept; // move constructor (4)  
List& operator=( List&& aOther ) noexcept; // move assignment (8)  
void swap( List& aOther ) noexcept; // swap elements (9)  
// basic operations  
size\_t size() const noexcept; // list size (2)  
template<typename U>  
void push\_front( U&& aData ); // add element at front (24)  
template<typename U>  
void push\_back( U&& aData ); // add element at back (24)  
void remove( const T& aElement ) noexcept; // remove element (36)  
const T& operator[]( size\_t aIndex ) const; // list indexer (14)  
// iterator interface  
Iterator begin() const noexcept; // (4)  
Iterator end() const noexcept; // (4)  
Iterator rbegin() const noexcept; // (4)  
Iterator rend() const noexcept; // (4)  
};  
The template class List defines an “object adapter” for DoublyLinkedList objects (i.e.,  
the list representation). There are three parts to the implementation: the basic list features,  
the copy semantics, and the move semantics.

**Problem 1**  
Implement the basic list features.  
The default list is empty. The method size() just returns the number of list elements.  
The create new list elements you need to use DoublyLinkedNode’s makeNode() method.  
For push\_front() and push\_back() to work properly, we need to use perfect forwarding  
when calling makeNode(). In addition, you need to insert the new element properly into the  
doubly-linked chain and update fHead and fTail (the list endpoints). Finally, adding an  
element to the list increases its size.  
Remove() deletes the first match from the list. If the list does not contain the element in  
question, the list remains unchanged. The element to be removed, once found, must be  
isolated. In addition, it may be necessary to adjust fHead and fTail when the removed  
element is the first and last element, respectively. Please note, the removed element is  
automatically destructed when it goes out of scope.  
The indexer has to return the corresponding element, if the index is within bounds. The first  
element in the list has index 0. The last element has index size() – 1.  
The iterator methods return the corresponding iterators. See tutorial 10 for details of how to  
obtain the corresponding iterators.  
To test your implementation of the basic features, uncomment #define P1 and compile  
your solution.  
The test driver should produce the following output:  
Test basic list functions:  
List size: 6  
5th element: eeee  
Remove 5th element.  
New 5th element: ffff  
List size: 5  
Forward iteration:  
aaaa  
bbbb  
cccc  
dddd  
ffff  
Backwards iteration:  
ffff  
dddd  
cccc  
bbbb  
aaaa  
Test basic list functions complete.

Initial code:

// List.h

#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**; // default constructor (2)

// copy semantics

List( **const** List& aOther ); // copy constructor (10)

List& **operator**=( **const** List& aOther ); // copy assignment (14)

// move semantics

List( List&& aOther ) **noexcept**; // move constructor (4)

List& **operator**=( List&& aOther ) **noexcept**; // move assignment (8)

**void** swap( List& aOther ) **noexcept**; // swap elements (9)

// basic operations

size\_t size() **const** **noexcept**; // list size (2)

**template**<**typename** U>

**void** push\_front( U&& aData ); // add element at front (24)

**template**<**typename** U>

**void** push\_back( U&& aData ); // add element at back (24)

**void** remove( **const** T& aElement ) **noexcept**; // remove element (36)

**const** T& **operator**[]( size\_t aIndex ) **const**; // list indexer (14)

// iterator interface

Iterator begin() **const** **noexcept**; // (4)

Iterator end() **const** **noexcept**; // (4)

Iterator rbegin() **const** **noexcept**; // (4)

Iterator rend() **const** **noexcept**; // (4)

};

Chat Prompt:  
Using the below Specifications to and extend the code file List.h following all specifications given, you will also being provided the reference files List.h (initial List.h file that you need to implement additionally), DoublyLinkedList.h, DoublyLinkedListIterator.h and Main.cpp (test file) to support your response.

1. Main.cpp (test file):

std::cout << "Test basic list functions:" << std::endl;

std::string lB( "bbbb" );

std::string lD( "dddd" );

lList.push\_back( lD );

lList.push\_front( "cccc" );

lList.push\_back( "eeee" );

lList.push\_front( lB );

lList.push\_back( "ffff" );

lList.push\_front( "aaaa" );

std::cout << "List size: " << lList.size() << std::endl;

std::cout << "5th element: " << lList[4] << std::endl;

lList.remove( lList[4] );

std::cout << "Remove 5th element." << std::endl;

std::cout << "New 5th element: " << lList[4] << std::endl;

std::cout << "List size: " << lList.size() << std::endl;

std::cout << "Forward iteration:" << std::endl;

for ( auto& item : lList )

{

std::cout << item << std::endl;

}

std::cout << "Backwards iteration:" << std::endl;

for ( auto iter = lList.rbegin(); iter != iter.rend(); iter-- )

{

std::cout << \*iter << std::endl;

}

std::cout << "Test basic list functions complete." << std::endl;

2. List.h, you need to implement this file as specifications given, therefore, allowing Main file printing accurate result.

#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; // default constructor (2)

// copy semantics

List( const List& aOther ); // copy constructor (10)

List& operator=( const List& aOther ); // copy assignment (14)

// move semantics

List( List&& aOther ) noexcept; // move constructor (4)

List& operator=( List&& aOther ) noexcept; // move assignment (8)

void swap( List& aOther ) noexcept; // swap elements (9)

// basic operations

size\_t size() const noexcept; // list size (2)

template<typename U>

void push\_front( U&& aData ); // add element at front (24)

template<typename U>

void push\_back( U&& aData ); // add element at back (24)

void remove( const T& aElement ) noexcept; // remove element (36)

const T& operator[]( size\_t aIndex ) const; // list indexer (14)

// iterator interface

Iterator begin() const noexcept; // (4)

Iterator end() const noexcept; // (4)

Iterator rbegin() const noexcept; // (4)

Iterator rend() const noexcept; // (4)

};

3. DoublyLinkedList.h:

#pragma once

#include <memory>

#include <algorithm>

template<typename T>

class DoublyLinkedList

{

public:

using Node = std::shared\_ptr<DoublyLinkedList<T>>;

using Next = std::shared\_ptr<DoublyLinkedList<T>>;

using Previous = std::weak\_ptr<DoublyLinkedList<T>>;

T fData;

Node fNext;

Previous fPrevious;

// factory method for list nodes

template<typename... Args>

static Node makeNode( Args&&... args )

{

// make\_share<T, Args...>

return std::make\_shared<DoublyLinkedList>( std::forward<Args>(args)... );

}

DoublyLinkedList( const T& aData ) noexcept :

fData(aData),

fNext(),

fPrevious()

{}

DoublyLinkedList( T&& aData ) noexcept :

fData(std::move(aData)),

fNext(),

fPrevious()

{}

void isolate() noexcept

{

if ( fNext )

{

fNext->fPrevious = fPrevious;

}

Node lNode = fPrevious.lock();

if ( lNode )

{

lNode->fNext = fNext;

}

fPrevious.reset();

fNext.reset();

}

void swap( DoublyLinkedList& aOther ) noexcept

{

std::swap( fData, aOther.fData );

}

};

4. DoublyLinkedListIterator.h:

#pragma once

#include <cassert>

#include "DoublyLinkedList.h"

template<typename T>

class DoublyLinkedListIterator

{

public:

using Iterator = DoublyLinkedListIterator<T>;

using Node = typename DoublyLinkedList<T>::Node;

enum class States

{

BEFORE,

DATA,

AFTER

};

DoublyLinkedListIterator( const Node& aHead, const Node& aTail ) noexcept :

fHead(aHead),

fTail(aTail),

fCurrent(aHead),

fState(States::DATA)

{

// sound head and tail hooks

assert( !fHead == !fTail );

if ( !fHead )

{

fState = States::AFTER;

}

}

const T& operator\*() const noexcept

{

return fCurrent->fData;

}

Iterator& operator++() noexcept // prefix

{

switch ( fState )

{

case States::BEFORE:

fCurrent = fHead;

if ( fCurrent )

{

fState = States::DATA;

}

else

{

fState = States::AFTER;

}

break;

case States::DATA:

fCurrent = fCurrent->fNext;

if ( !fCurrent )

{

fState = States::AFTER;

}

break;

case States::AFTER:

break;

}

return \*this;

}

Iterator operator++(int) noexcept // postfix

{

Iterator old = \*this;

++(\*this);

return old;

}

Iterator& operator--() noexcept // prefix

{

switch ( fState )

{

case States::BEFORE:

break;

case States::DATA:

fCurrent = fCurrent->fPrevious.lock();

if ( !fCurrent )

{

fState = States::BEFORE;

}

break;

case States::AFTER:

fCurrent = fTail;

if ( fCurrent )

{

fState = States::DATA;

}

else

{

fState = States::BEFORE;

}

break;

}

return \*this;

}

Iterator operator--(int) noexcept // postfix

{

Iterator old = \*this;

--(\*this);

return old;

}

bool operator==( const Iterator& aOther ) const noexcept

{

return

fHead == aOther.fHead &&

fTail == aOther.fTail &&

fState == aOther.fState &&

fCurrent == aOther.fCurrent;

}

bool operator!=( const Iterator& aOther ) const noexcept

{

return !(\*this == aOther);

}

Iterator begin() const noexcept

{

return ++(rend());

}

Iterator end() const noexcept

{

Iterator iter = \*this;

iter.fCurrent = nullptr;

iter.fState = States::AFTER;

return iter;

}

Iterator rbegin() const noexcept

{

return --(end());

}

Iterator rend() const noexcept

{

Iterator iter = \*this;

iter.fCurrent = nullptr;

iter.fState = States::BEFORE;

return iter;

}

private:

Node fHead;

Node fTail;

Node fCurrent;

States fState;

};

After revising the file provided, you have to examine these Specifications criteria and implement the List.hfile as instructed:

Using the template classes DoublyLinkedList and DoublyLinkedListIterator, implement the template class List as specified below:

#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; // default constructor (2)

// copy semantics

List( const List& aOther ); // copy constructor (10)

List& operator=( const List& aOther ); // copy assignment (14)

// move semantics

List( List&& aOther ) noexcept; // move constructor (4)

List& operator=( List&& aOther ) noexcept; // move assignment (8)

void swap( List& aOther ) noexcept; // swap elements (9)

// basic operations

size\_t size() const noexcept; // list size (2)

template<typename U>

void push\_front( U&& aData ); // add element at front (24)

template<typename U>

void push\_back( U&& aData ); // add element at back (24)

void remove( const T& aElement ) noexcept; // remove element (36)

const T& operator[]( size\_t aIndex ) const; // list indexer (14)

// iterator interface

Iterator begin() const noexcept; // (4)

Iterator end() const noexcept; // (4)

Iterator rbegin() const noexcept; // (4)

Iterator rend() const noexcept; // (4)

};

The template class List defines an “object adapter” for DoublyLinkedList objects (i.e.,

the list representation). There are three parts to the implementation: the basic list features,

the copy semantics, and the move semantics.

Problem 1

Implement the basic list features.

The default list is empty. The method size() just returns the number of list elements.

The create new list elements you need to use DoublyLinkedNode’s makeNode() method.

For push\_front() and push\_back() to work properly, we need to use perfect forwarding

when calling makeNode(). In addition, you need to insert the new element properly into the

doubly-linked chain and update fHead and fTail (the list endpoints). Finally, adding an

element to the list increases its size.

Remove() deletes the first match from the list. If the list does not contain the element in

question, the list remains unchanged. The element to be removed, once found, must be

isolated. In addition, it may be necessary to adjust fHead and fTail when the removed

element is the first and last element, respectively. Please note, the removed element is

automatically destructed when it goes out of scope.

The indexer has to return the corresponding element, if the index is within bounds. The first

element in the list has index 0. The last element has index size() – 1.

The iterator methods return the corresponding iterators. See tutorial 10 for details of how to

obtain the corresponding iterators.

To test your implementation of the basic features, uncomment #define P1 and compile

your solution.

The test driver should produce the following output:

Test basic list functions:

List size: 6

5th element: eeee

Remove 5th element.

New 5th element: ffff

List size: 5

Forward iteration:

aaaa

bbbb

cccc

dddd

ffff

Backwards iteration:

ffff

dddd

cccc

bbbb

aaaa

Test basic list functions complete.