Swinburne University of Technology

School of Science, Computing and Engineering Technologies

ASSIGNMENT COVER SHEET

Subject Code: COS30008
Subject Title: Data Structures and Patterns
Assignment number and title: 3, List ADT
Due date: Monday, May 15, 2023, 10:30
Lecturer: Dr. Markus Lumpe

Your name: Brandy Dan **Your student id:** 103864526

Check Tutorial	Tues 08:30	Tues 10:30	Tues 12:30 BA603	Tues 12:30 ATC627	Tues 14:30	Wed 08:30	Wed 10:30	Wed 12:30	Wed 14:30	Thurs 08:30	Thurs 10:30
						X					

Marker's comments:

Problem	Marks	Obtained
1	118	
2	24	
3	21	
Total	163	

Extension certification:

This assignment has been given an extension and is now due on _____

Signature of Convener: _____

```
// COS30008, Problem Set 3, 2023

#pragma once

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


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

   Node fHead;
   Node fTail;
   size_t fSize;

public:

   using Iterator = DoublyLinkedListIterator<T>;

   List() noexcept :
      fHead(),
      fTail(),
      fSize(0)
   {}


   List(const List& aOther):
      fHead(aOther.fHead),
      fTail(aOther.fTail),
      fSize(aOther.fSize)

   {}

   List& operator=(const List& aOther)
   {
      if (this != &aOther)
      {
         new (this) List(aOther);
      }

      return *this;
   }


   List(List&& aOther) noexcept :
      List()
   {
      swap(aOther);
   }
```

```
List& operator=(List&& aOther) noexcept
{
   if (this != &aOther)
   {
      swap(aOther);
   }

   return *this;
}

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


size_t size() const noexcept
{
   return fSize;
}

template<typename U>
void push_front(U&& aData)
{

   if (fHead)
   {
      Node lnode = DoublyLinkedList<T>::makeNode(std::forward<U>(aData));

      lnode->fNext = fHead;
      fHead->fPrevious = lnode;
      fHead = lnode;

   }
   else
   {
      fHead = DoublyLinkedList<T>::makeNode(std::forward<U>(aData));
      fTail = fHead;

   }

   fSize++;

}

template<typename U>
void push_back(U&& aData)
{
   if (fTail)
   {
      Node lnode = DoublyLinkedList<T>::makeNode(std::forward<U>(aData));
```

```
        lnode->fPrevious = fTail;
        fTail->fNext = lnode;
        fTail = lnode;

    }
    else
    {
        fTail = DoublyLinkedList<T>::makeNode(std::forward<U>(aData));
        fHead = fTail;
    }

    fSize++;

}

void remove(const T& aElement) noexcept
{
    Iterator lIter(fHead, fTail);

    Node current = fHead;
    bool found = false;

    for (auto& item : lIter)
    {

        if (item == aElement)
        {
            fSize--;
            found = true;
            break;
        }

        current = current->fNext;

    }

    if (found)
    {
        if (current == fHead)
        {
            fHead = fHead->fNext;



        }
        else if (current == fTail)
        {
            fTail = fTail->fPrevious.lock();

        }

        current->isolate();

    }
```

```
    }

    const T& operator[](size_t aIndex) const
    {
        assert(aIndex < fSize);
        Node current = fHead;

        for (size_t i = 0; i < aIndex; i++)
        {
            current = current->fNext;

        }
        return current->fData;

    }


    Iterator begin() const noexcept
    {
        return Iterator(fHead, fTail).begin();
    }
    Iterator end() const noexcept
    {
        return Iterator(fHead, fTail).end();
    }
    Iterator rbegin() const noexcept
    {
        return Iterator(fHead, fTail).rbegin();
    }
    Iterator rend() const noexcept
    {
        return Iterator(fHead, fTail).rend();
    }
};
```

```
// COS30008, Tutorial 10, 2023

#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 );
   }
};
```

```
// COS30008, Tutorial 10, 2023

#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;
};
```

```
// COS30008, Problem Set 3, 2023

#include <iostream>
#include <string>

#include "List.h"

#define P1
#define P2
#define P3

int main()
{
   using StringList = List<std::string>;

   StringList lList;

#ifdef P1

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

   lList.push_back( "DDDD" );
   lList.push_front( "CCCC" );
   lList.push_back( "EEEE" );
   lList.push_front( "BBBB" );
   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;

#endif

#ifdef P2

    std::cout << "Test copy semantics:" << std::endl;

    StringList lCopy = lList;

    std::cout << "Copied list iteration (source):" << std::endl;

    for ( auto& item : lList )
    {
        std::cout << item << std::endl;
    }

    std::cout << "Copied list iteration (target):" << std::endl;

    for ( auto& item : lCopy )
    {
        std::cout << item << std::endl;
    }

    std::cout << "Test copy semantics complete." << std::endl;

#endif

#ifdef P3

    std::cout << "Test move semantics:" << std::endl;

    StringList lMoveCopy = std::move( lList );

    std::cout << "Moved list iteration (source):" << std::endl;

    for ( auto& item : lList )
    {
        std::cout << item << std::endl;
    }

    std::cout << "Moved list iteration (target):" << std::endl;

    for ( auto& item : lMoveCopy )
    {
        std::cout << item << std::endl;
    }

    std::cout << "Test move semantics complete." << std::endl;

#endif
```

```
#ifndef P1

    #ifndef P2

        #ifndef P3

        std::cout << "No test enabled." << std::endl;

        #endif

    #endif
#endif

    return 0;
}
```