1. **Requirements Documentation** 
   1. **Description of Problem**

**Name:** Linked List

**Problem Statement:** (Create a Linked List and demonstrate it working error free.)

**Problem Specifications:** (Linked List classes completed to U.M.L specs.)

1. **System Architecture**
2. **Linked List Iterator. h**

**Prototype:** (<Type> operator\*)

**Description:** ()

**Precondition:** (What is needed for the function to start execution)

**Post condition:** (What is the condition of the application after function execution)

**Protection Level:** (public)

**Prototype:** (bool operator= =(const Linked List Type<type>&other)

**Description:** (overload the equal operator)

**Post condition:** (returns true if iterator is equal otherwise return false.)

**Protection Level:** (public)

**Prototype:** (bool operator !=( const Linked List Type<type>&other)

**Description:** (evaluates iterator)

**Post condition:** (returns true if iterator is not equal otherwise return false)

**Protection Level:** (public)

**Prototype:** (Linked List Iterator<Type>operator++())

**Description:** (a function that advances the next node)

**Post condition:** ((the iterator is advanced to the next node.)

**Protection Level:** (public)

**Link List Type .h**

Prototype: (void Copy List (const Linked List Type<Type>&other list)

Description: (a function that copies one list and returns the copy as a new list)

Post condition: (What is the condition of the application after function execution)

Protection Level: (Private)

Prototype: (const Linked List Type<Type>&operator= (const Linked List Type &other)

Description: (a function that assigns the info in one node to be the same as the entered second node)

Precondition: (Node must not be Null)

Post condition: (copy of node is returned in another Node)

Protection Level: (Public)

Prototype: (void Initialize List ())

Description: (initialize the list to an empty state.)

Precondition: (list must not be null)

Post condition: (first=NULL count =0)

Protection Level: (Public)

Prototype: (bool is Empty () const)

Description: (conditional statement that tells if the list empty.)

Precondition: (check if first is NULL)

Post condition: (if first is NULL return true. //else if first is not NULL return false.)

Protection Level: (Public)

Prototype: (void Print() const)

Description: (a function that prints out the if of the list and the nodes in the list)

Precondition: (there must be info to print out)

Post condition: (copy of node is returned in another Node)

Protection Level: (Public)

Prototype: (int Length())

Description: (a function that return the number of nodes in the list)

Precondition: (Node must not be Null)

Post condition: (the amount of nodes in the list is returned)

Protection Level: (Public)

Prototype: (void Destroy List())

Description: (function to delete all the nodes from the list.)

Precondition: (Node must not be Null)

Post condition: (all of the nodes in the list are destroyed)

Protection Level: (Public)

Prototype: (Type Front() const)

Description: (Function to return the last element of the list.)

Precondition: (List must exist and must not be empty)

Post condition: (the last element of the list is returned)

Protection Level: (Public)

Prototype: (Type Back() const )

Description: (Function to return the first element of the list.)

Precondition: (List must exist and must not be empty)

Post condition: (the first element of the list is returned)

Protection Level: (Public)

Prototype: (virtual bool Search (const Type & other) const)

Description: (// A function that searches the list for a match and returns true if a match is found and false if a match is not found.)

Precondition: (List must exist and must not be empty)

Post condition: (returns true if a match is found and false if a match is not found.)

Protection Level: (Public)

Prototype: (virtual void Insert First (const Type & other))

Description: (a function that adds a new first node to the list)

Precondition: (List must exist and must not be empty)

Post condition: the new node is the new first node)

Protection Level: (Public)

Prototype: (virtual void Insert Last (const Type& other))

Description: (a function that adds a new last node to the list)

Precondition: (List must exist and must not be empty)

Post condition: (the new node is the new last node)

Protection Level: (Public)

Prototype: (Type Begin (Linked List Iterator<Type>&other))

Description: (a function that indicates the beginning of the lis)

Precondition: (List must exist and must not be empty)

Post condition: (the first node is set to the beginning of the list)

Protection Level: (Public)

Prototype: (Type End (Linked List Iterator<Type>&other))

Description: (a function that indicates the ending of the list)

Precondition: (List must exist and must not be empty)

Post condition: (the last node is set to the end of the list)

Protection Level: (Public)

**Source Code**

(Copy paste code from Visual Studios)

**Linked List Iterator. h**

#pragma once

#include "NodeType.h"

template <class Type>

class LinkedListIterator

{

private:

NodeType<Type>\*current;

public:

LinkedListIterator()// sets current equal to NULL.

{

current = NULL;

};

LinkedListIterator(NodeType<Type>\*ptr)//Sets current equal to ptr.

{

current = ptr;

};

LinkedListIterator<Type>operator++()//the iterator is advanced to the next node.

{

return this ->current = current->link;

}

//function to overload the prre-increment operator.

//post condition the iterator is advanced to the next node.

LinkedListIterator<Type> \*operator\*()

{

return this->current->info;

};

bool operator==(const LinkedListIterator<Type>&right)const//return true if iterator is equal.//by right otherwise it returns false.

{

return true this->current == right->current;

}

bool operator!=(const LinkedListIterator<Type>&right)const//return true if the iterator is not equal.//by right otherwise it returns false.

{

return true this->current != right->current;

}

};

**LinkedListType .h**

#pragma once

#include <iostream>

#include"NodeType.h"

#include "LinkedListsIterator.h"

template < class Type>

class LinkedListType

{

protected:

int count;

NodeType<Type>\*first;

NodeType<Type>\*last;

private:

void CopyList(const LinkedListType<Type>&otherlist)

{

NodeType<Type>\*tempNode;

NodeType<Type>\*current;

tempNode = current;

current = otherlist.first;

first = tempNode;

last = tempNode;

InsertFirst(current->info);

current = current->link;

while(current!='\0')

{

InsertLast(current->info);

current = current->link;

}

}

public:

const LinkedListType<Type>&operator=(const LinkedListType&other)

{

if (first == other.first)

{

CopyList();

}

return first;

};

void InitilizeList()//intialize the list to an empty state.//postcondition first=NULL count =0.//postcondition first=NULL count =0.

{

while(first!=NULL)

DestroyList();

};

bool isEmpty()const//conditional statement that tells if the list empty.//check if first is NULL//if first is NULL return true.//else if first is not NULL return false.

{

if (first != '\0')

{

return true;

}

else (first != '\0');

{

return false;

}

};

void Print()const//function to return the nodes in the list.

{

NodeType<Type> \*current=first;

if(current!=NULL)

{

}

current->link->info;

std::cout << current->info<<std::endl;

current = current->link;

};

int Lenght()//iterates through list.//returns the amount of nodes in the list.

{

return count;

};

//function to delete all the nodes from the list.

//postcondition first =NULL count =0.

//made a temp node.

//used a while for while first is not equal to last.

//referenced the temp node to assign the temp to first.

//then assigned first to link.

//then delete the referenced temp.

void DestroyList()

{

NodeType<Type>\*current;

while (first != NULL)

{

current = first;

first = first->link;

delete current;

}

last = '\0';

count = 0;

};

//Function to return the last element of the list.

//precondition the list must exist and must not be empty

//post condition if the list is empty

Type Front()const

{

(first != '\0');

return first;

}

//Function to return the first element of the list.

//precondition the list must exist and must not be empty

//post condition if the list is empty

Type Back()const

{

(last!='\0');

return last;

}

virtual bool Search(const Type&other)const// A function that searches the list for a match and returns true if a match is found and false if a match is not found.

{

NodeType<Type> \*result=new NodeType<Type>;

while(first !='\0')

{

result = first->link;

if(result->info!=other)

{

return true;

}

return false;

}

}

virtual void InsertFirst( const Type&other)// a function that adds a new first node to the list

{

NodeType<Type> \*tempNode = new NodeType<Type>;

tempNode->info = other;

tempNode->link = first;

first = tempNode;

if (count==0)

{

last = first;

}

count++;

}

virtual void InsertLast(const Type& other)// a function that adds a new last node to the list

{

NodeType<Type> \*tempNode = new NodeType<Type>;

tempNode->info = other;

last->link = tempNode;

last=tempNode;

last->link = '\0';

if (count == 0)

{

first = last;

}

count++;

}

Type Begin(LinkedListIterator<Type>&other)// a function that indacates the begining of the list

{

LinkedListIterator<Type> \*tempNode;

tempNode->current = first;

return first;

}

Type End(LinkedListIterator<Type>&other)// a functin that indacates the endding of the list

{

LinkedListIterator<Type>\*tempNode;

tempNode->current = last;

return last;

}

};

Nodetype. h

#pragma once

template<typename Type>

struct NodeType

{

Type info;

NodeType<Type>\*link;

};

1. **Read Me**

(Be very clear as to how the assessor should go about getting your application, running it, and using it. You should assume the assessor knows nothing about your application.)