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:** (A function that returns the info in a node.)

**Protection Level:** (public)

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

**Description:** (Returns true if this iterator is equal to the iterator specified by right otherwise it returns false.)

**Protection Level:** (public)

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

**Description:** (Returns true if this iterator is not equal to the iterator specified by right//otherwise it returns false)

**Protection Level:** (public)

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

**Description:** (The iterator is advanced to 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 "Node type. h"

#include <cstddef>

template<class Type>

class Link List Iterator

{

private:

//Pointer to point to the current node in the linked list

Node Type<Type> \*current;

public:

//sets current == NULL

Link List Iterator()

{

current = NULL;

}

//sets current = pointer.

Link List Iterator(Node Type<Type> \*pointer)

{

current = pointer;

}

//returns the info contained in the node

Type operator\*()

{

return current->info;

}

//the iterator is advanced to the next node

Link List Iterator<Type>operator++()

{

current = current->link;

return \*this;

}

//returns true if this iterator is equal to the iterator specified by right

//otherwise it returns false

bool operator==(const Link List Iterator<Type>& right) const

{

return current == right->current;

}

//returns true if this iterator is not equal to the iterator specified by right

//otherwise it returns false

bool operator!=(const Link List Iterator<Type>& right) const

{

return current != right->current;

}

};

**LinkedListType .h**

#pragma once

#include <iostream>

#include "Node Type. h"

#include "Linked List Iterator.h"

#include <assert.h>

template<class Type>

class Link List Type

{

protected:

int count;

Node Type<Type> \*first;

Node Type<Type> \*last;

public:

//Overload the assignment

const Link List Type<Type>& operator=(const Link List Type<Type>& other)

{

if (first == other. first)

Copy List();

else

return first;

}

//Returns the list to an empty state

//first = NULL, last = NULL, count = 0

void Initialize List()

{

while (first != NULL)

{

Destroy List();

}

}

//A function to determine if the list is empty

// returns true if the list is empty otherwise it returns false

bool Is Empty List()

{

return first == NULL;

}

// A function to return the of nodes in the list

void const Print()

{

Node Type<Type> \*temp = first;

while (temp != NULL)

{

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

temp = temp->link;

}

}

//function to return the length of the nodes in the list

int Length()

{

return count;

}

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

void Destroy Lsit()

{

NodeType<Type> \*temp;

while (first != NULL)

{

temp = first;

first = first->link;

delete temp;

}

last = NULL;

count = 0;

}

// A function to return the first element of the list

// they must exist and must not be empty

// if the list is empty, the program terminates; otherwise, the first

Type Front()

{

assert(first != NULL);

return first->info;

}

//function to return the last element of the list

// the must exist and must not be empty

//if the list is empty, the program terminates; otherwise, the last

//element of the list is returned

Type Back()

{

assert(last != NULL);

return last->info;

}

//function to determine whether search Item is on the list

//returns true if search Item is in the list, otherwise the value

//otherwise false is returned

bool Search(const Type& other) const

{

Node Type<Type> \*current = new Node Type<Type>;

while (first != NULL)

{

current = first->link;

if (current->info == other)

{

return true;

}

return false;

}

}

//function to insert new Item at the beginning of the list

//last points of the last node in the list, and count is increased by 1

void InsertFirst(const Type& other)

{

Node Type<Type> \*new Node = new Node Type<Type>;

new Node->info = other;

new Node->link = first;

first = new Node;

if (count == 0)

last = first;

count++;

}

//function to insert new Item at the end of the list

//Post condition: last points to the new list, new Item is inserted at the end

//of the list

//last points of the last node in the list, and count in incremented by 1

void Insert Last(const Type& other)

{

Node Type<Type> \*new Item = new Node Type<Type>;

New Item->info = other;

last->link = new Item;

last = new Item;

last->link = NULL;

if (count == 0)

first = last;

count++;

}

//function to delete Item from the list

//if found, the node containing delete Item from the list. First

// points to the first node, last points to the last node of the updated list,

//and counts is decreased by 1

void Delete Node(const Type& other)

{

Node Type<Type> \*current = new Node Type<Type>;

Node Type<Type> \*next = new Node Type<Type>;

if (first->info == other)

{

current = first;

first = first->link;

delete current;

count--;

return;

}

current = first->link;

while (current != NULL) {

if (current->link->info == other)

{

Node Type<Type> \*Node To Delete = current->link;

current->link = Node To Delete->link;

delete Node To Delete;

count--;

return;

}

current = current->link;

}

}

//function to return an iterator at the beginning of the linked list

Link List Iterator<Type> Begin()

{

Link List Iterator<Type> \*temp Node;

Temp Node->current = first;

return first;

}

//function to return an iterator at the end of the linked list

Link List Iterator<Type> End()

{

Link List Iterator<Type> \*temp Node;

Temp Node->current = last;

return last;

}

//Default Constructor

//Initializes the last to an empty state

LinkListType()

{

first = NULL;

last = NULL;

count = 0;

}

LinkListType(const LinkListType<Type>& other)

{

first = NULL;

CopyList();

}

//destructor

//deletes all the nodes from the list

~LinkListType()

{

delete first;

delete last;

count = 0;

}

private:

//function to make a copy of otherList

// a copy of otherList is created and assigned to this list

void CopyList(const LinkListType<Type>& other)

{

NodeType<Type>\*newTempNode;

NodeType<Type>\*current;

current = other.first;

newTempNode = current;

first = newTempNode;

last = newTempNode;

InsertFirst(current->info);

current = current->link;

while (current != NULL)

{

InsertLast(current->info);

current = current->link;

}

}

};

Nodetype. h

#pragma once

template<typename Type>

struct NodeType

{

Type info;

NodeType<Type>\*link;

};

**Main.cpp**

#include <iostream>

#include "LinkedListtype.h"

#include <assert.h>

int main()

{

LinkListType<int> \*listNumberOne = new LinkListType<int>();

LinkListType<int> \*listNumberTwo = new LinkListType<int>();

listNumberOne->InsertFirst(1);

listNumberOne->InsertFirst(2);

listNumberOne->InsertLast(3);

listNumberOne->Print();

listNumberTwo->InsertFirst(1);

listNumberTwo->InsertFirst(2);

listNumberTwo->InsertLast(3);

listNumberTwo->InsertLast(4);

listNumberTwo->Print();

assert(listNumberOne->Length() == 3);

assert(listNumberTwo->Length() == 4);

system("pause");

}

1. **Read Me**
2. (Go to my github look for the repository named RND\_1221LinkedList and download a zip file.)
3. (Once file has completed downloading go to downloads look for the file you just downloaded and extract all the files.)
4. (Once all files are all extracted go to the look for the solution file to open file in Microsoft visual studios.)
5. (Once file is opened in microsoft visual studios look for the debug button or press f5 to start application.)