**Donray Williams**

**12/14/16**

**Linked List– Assessment 2 – Year 1**

# **1.0 Requirements Documentation**

# **1.1 Description of problem**

**A. Name:** Linked List

**B. Problem Statement:** Create a Linked List and Linked List Iterator Class based off of the example that was given to us by Dylan Guidry and Matthew Williamson.

**C. Problem Specification:** The Linked List and Linked List Iterator must have the required functionality that was given to us in the example created by Dylan Guidry. It must be able to add nodes to the front and back, delete nodes or delete entire list and so forth.

**1.2 Input Information**

A. Not Applicable

**1.3 Output Information**

A. The console displays whatever function is being used to test.

**1.4 User Interface**

A. Not Applicable

# **2.0 System Architecture**

**2.1 Member Functions in the classes:**

**Class** linkedListType ~

**PROTOTYPE:** void initializeList ()

**DESCRIPTION:** Initialize the list to an empty state

**PRECONDITION:** Must have a list to be initialized

**POST CONDITION:** first = NULL, last = NULL, count = 0;

**VISIBILITY:** Public

**PROTOTYPE:** bool isEmptyList () const

**DESCRIPTION:** Function to determine whether the list is empty

**PRECONDITION:** There must be a list to check if it is empty

**POST CONDITION:** Returns true if the list is empty otherwise it returns false

**VISIBILITY:** Public

**PROTOTYPE:** const linkedListType<Type>& operator= (const linkedListType<Type>& otherList**)**

**DESCRIPTION:** Overload the assignment operator

**PRECONDITION:** Must have a = operator for this overload to ever be used

**POST CONDITION:** The two linked list are equal to each other

**VISIBILITY:** Public

**PROTOTYPE:** void print () const

**DESCRIPTION:** Function to output the data contained in each node

**PRECONDITION:** Must have data to be printed

**POST CONDITION:** Prints out the nodes in the list

**VISIBILITY:** Public

**PROTOTYPE:** bool search (const Type& nodeInfo)

**DESCRIPTION:** Function to determine whether node is in the list

**PRECONDITION:** Must have a list to search through

**POST CONDITION:** Returns true if node is in the list

otherwise the value false is returned

**VISIBILITY:** Public

**PROTOTYPE:** void destroyList ()

**DESCRIPTION:** Function to delete all the nodes from the list

**PRECONDITION:** Must have a list to delete

**POST CONDITION:** first = NULL, last = NULL, count = 0;

**VISIBILITY:** Public

**PROTOTYPE:** void deleteNode (const Type& nodeInfo)

**DESCRIPTION:** Function to delete node from the list

**PRECONDITION:** The node must be there to delete

**POST CONDITION:** If found, the node containing the node is deleted from the list. First points to the first node, last points to the last node of the update list, and count is decremented by 1

**VISIBILITY:** Public

**PROTOTYPE:** ~linkedListType<Type> ()

**DESCRIPTION:** Destructor, calls the destroyList () function

**PRECONDITION:** Must have nodes to delete

**POST CONDITION:** The list object is destroyed

**VISIBILITY:** Public

**PROTOTYPE:** void insertLast (const Type& node)

**DESCRIPTION:** Function to insert node at the end of the list

**PRECONDITION:** Need a list to start off with

**POST CONDITION:** First points to the new list, node is inserted

at the beginning of the list, last points to the last node in

the list, and count is incremented by 1.

**VISIBILITY:** Public

**PROTOTYPE:** linkedListType<Type> (const linkedListType<Type> & otherList)

**DESCRIPTION:** Copy constructor

**PRECONDITION:** Must have list to copy

**POST CONDITION:** Constructor is copied

**VISIBILITY:** Public

**PROTOTYPE:** void insertFirst (const Type& node)

**DESCRIPTION:** Function to insert node at the beginning of the list

**PRECONDITION:** Must have a list

**POST CONDITION:** First points to the new list, node is inserted

at the beginning of the list, last points to the last node in

the list, and count is incremented by 1.

**VISIBILITY:** Public

**PROTOTYPE:** Type front () const

**DESCRIPTION:** Function to return the first element in the list

**PRECONDITION:** The list must exist and must not be empty

**POST CONDITION:** If the list is empty, the program terminates; otherwise,

the first element of the list is returned.

**VISIBILITY:** Public

**PROTOTYPE:** Type back () const

**DESCRIPTION:** Function to return the last element in the list

**PRECONDITION:** The list must exist and must not be empty

**POST CONDITION:** If the list is empty, the program terminates; otherwise,

the last element of the list is returned.

**VISIBILITY:** Public

**PROTOTYPE:** linkedListIterator<Type> begin ()

**DESCRIPTION:** Function to return an iterator at the beginning of the linked list

**PRECONDITION:** Must have a list

**POST CONDITION:** Returns an iterator such that the current is set to first

**VISIBILITY:** Public

**PROTOTYPE:** linkedListIterator<Type> end ()

**DESCRIPTION:** Function to return an iterator at the end of the linked list

**PRECONDITION:** Must have a list

**POST CONDITION:** Returns an iterator such that current is set to NULL

**VISIBILITY:** Public

**PROTOTYPE:** const int length ()

**DESCRIPTION:** Function to return the number of nodes in the list

**PRECONDITION:** List must exist

**POST CONDITION:** The value of count is returned

**VISIBILITY:** Public

**PROTOTYPE:** void copyList (const linkedListType<Type>& otherList)

**DESCRIPTION:** Function to make a copy of list

**PRECONDITION:** Must have a list

**POST CONDITION:** A copy of list is created and assigned to this list

**VISIBILITY:** Private

**Class** linkedListIterator ~

**PROTOTYPE:** Type operator \*()

**DESCRIPTION:** Function to overload the dereferencing operator \*

**PRECONDITION:** Must have two list to check against

**POST CONDITION:** Returns the info contained in the node

**VISIBILITY:** Private

**PROTOTYPE:** linkedListIterator<Type> operator++ ()

**DESCRIPTION:** Overload the pre-increment operator

**PRECONDITION:** Must have two list to check against

**POST CONDITION:** The iterator is advanced to the next node

**VISIBILITY:** Private

**PROTOTYPE:** bool operator = = (const linkedListIterator<Type>& a)

**DESCRIPTION:** Overload the equality operator

**PRECONDITION:** Must have two list to check against

**POST CONDITION:** Returns true if this iterator is equal to the

iterator specified by right otherwise returns false

**VISIBILITY:** Private

**PROTOTYPE:** bool operator! = (const linkedListIterator<Type>& a)

**DESCRIPTION:** Overload the not equal operator

**PRECONDITION:** Must have two list to check against

**POST CONDITION:** Returns true if this iterator is not equal to the

iterator specified by right otherwise returns false

**VISIBILITY:** Private

**Member Variables:**

**Struct** nodeType:

**int** *info*– Integer that stores the info of the list

**nodeType<Type> \*** *link –* Creates a nodeType pointer that will be the link between the list

**Class** linkedListType:

**Protected:**

**int** *count-* Integer that keeps track of the number of nodes going into the list

**nodeType<Type> \*** *first-* First pointer to the first node in the list

**nodeType<Type>** \* *last-* Last pointer to the last node of the list

**Class** linkedListIterator:

**Private:**

**nodeType<Type>** \* *current-* Current pointer to the current node in the list

Source Code:

#pragma once

#include <cassert>

template<class Type>

class linkedListIterator;

template<class Type>

struct nodeType

{

public:

int info;

nodeType<Type> \* link;

};

template<class Type>

class linkedListType

{

public:

linkedListType() {

first = new nodeType<Type>;

last = new nodeType<Type>;

count = 0;

};

//Initialize the list to an empty state

//Postcondition: first = NULL, last = NULL, count = 0;

void initializeList()

{

first = NULL;

last = NULL;

count = 0;

}

//Function to determine whether the list is empty

//Postcondition: Returns true if the list is empty otherwise it returns false

bool isEmptyList() const

{

return (first == nullptr) ? true : false;

}

//Overload the assignment operator

const linkedListType<Type>& operator= (const linkedListType<Type>& otherList)

{

copyList(otherList);

return \*this;

}

//Function to output the data contained in each node

//Postcondition: Node

void print() const

{

nodeType<Type> \*current;

current = first;

while (current != NULL)

{

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

current = current->link;

}

}

//Function to determine whether node is in the list

//Postcondition: Returns true if node is in the list

//otherwise the value false is returned

bool search(const Type& nodeInfo)

{

linkedListIterator<Type> tmp = linkedListIterator<Type>(begin());

while (\*tmp != NULL)

{

if (\*tmp == nodeInfo)

{

return true;

}

++tmp;

}

return false;

}

//Function to delete all the nodes from the list

//Postcondition: first = NULL, last = NULL, count = 0;

void destroyList()

{

nodeType<Type> \* iter = first;

while (iter != NULL)

{

nodeType<Type> \* tmp = iter;

iter = iter->link;

delete tmp;

}

first = NULL;

last = NULL;

count = 0;

}

//Function to delete node from the list

//Postcondition: If found, the node containing the node is deleted from the list. First points to

//the first node, last points to the last node of the update list, and count is decremented by 1

void deleteNode(const Type& nodeInfo)

{

nodeType<Type>\* node = first;

if (node == nullptr)

{

return;

}

else if (node->info == nodeInfo)

{

nodeType<Type> \*tmp = node;

node = node->link;

delete tmp;

count--;

first = node;

}

else

{

while (node->link != NULL)

{

if (node->link->info == nodeInfo)

{

nodeType<Type>\* tmp = node->link;

node->link = node->link->link;

delete tmp;

count--;

break;

}

node = node->link;

}

}

}

//Deconstructor

//Deletes all the nodes from the list

//Postcondition: The list object is destroyed

~linkedListType<Type>()

{

destroyList();

}

//Function to insert node at the end of the list

//Postcondition: first points to the new list, node is inserted

//at the beginning of the list, last points to the last node in

//the list, and count is incremented by 1;

void insertLast(const Type& node)

{

nodeType<Type> \* newNode;

newNode = new nodeType<Type>;

if (count == 0)

{

first->info = node;

first->link = nullptr;

last->info = node;

last->link = nullptr;

count++;

}

else

{

last->link = newNode;

last = newNode;

if (count == 1)

first->link = newNode;

last->info = node;

last->link = nullptr;

count++;

}

}

//Copy constructor

linkedListType<Type>(const linkedListType<Type> & otherList)

{

copyList(otherList);

}

//Function to insert node at the beginning of the list

//Postcondition: first points to the new list, node is inserted

//at the beginning of the list, last points to the last node in

//the list, and count is incremented by 1;

void insertFirst(const Type& node)

{

nodeType<Type> \* newNode;

newNode = new nodeType<Type>;

if (count == 0)

{

first->info = node;

first->link = nullptr;

last->info = node;

last->link = nullptr;

count++;

}

else

{

newNode->link = first;

first = newNode;

first->info = node;

count++;

}

}

//Function to return the first element in the list

//Precondition: The list must exist and must not be empty

//Postcondition: If the list is empty, the program terminates; otherwise,

//the first element of the list is returned

Type front()const

{

assert(count != 0);

return last->info;

}

//Function to return the last element in the list

//Precondition: The list must exist and must not be empty

//Postcondition: If the list is empty, the program terminates; otherwise,

//the last element of the list is returned

Type back()const

{

assert(count != 0);

return first->info;

}

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

//Postcondition: Returns an iterator such that the current is set to first

linkedListIterator<Type> begin()

{

return linkedListIterator<Type>(first);

}

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

//Postcondition: Returns an iterator such that current is set to NULL

linkedListIterator<Type> end()

{

return linkedListIterator<Type>(last);

}

//Function to return the number of nodes in the list

//Postcondition: The value of count is returned

const int length()

{

return count;

}

protected:

int count; //Variable to store the number of elements in the list

nodeType<Type> \* first; //Pointer to the first node of the list

nodeType<Type> \* last; //Pointer to the last node of the list

private:

//Function to make a copy of list

//Postcondition: A copy of list is created and assigned to this list

void copyList(const linkedListType<Type>& otherList)

{

first = otherList.first;

last = otherList.last;

count = otherList.count;

}

};

template<class Type>

class linkedListIterator

{

private:

nodeType<Type> \*current;

public:

linkedListIterator() {};

linkedListIterator(nodeType<Type> \*node) : current(node) {};

//Function to overload the dereferencing operator \*

//Postcondition: Returns the info contained in the node

Type operator \*()

{

if (current == nullptr)

return NULL;

return current->info;

}

//Overload the pre-increment operator

//Postcondition: The iterator is advanced to the next node

linkedListIterator<Type> operator++ ()

{

current = current->link;

return \*this;

}

//Overload the equality operator

//Postcondition: Returns true if this iterator is equal to the

//iterator specified by right otherwise returns false

bool operator ==(const linkedListIterator<Type>& a)

{

return current == a.current;

}

//Overload the not equal operator

//Postcondition: Returns true if this iterator is not equal to the

//iterator specified by right otherwise returns false

bool operator !=(const linkedListIterator<Type>& a)

{

return current != a.current;

}

};

**-Read Me –**

You can access this file via this link: <https://github.com/wdonray/Intro_To_Programming/tree/master/Linked>

Click the button that says “Clone or download” button listed on the top right in this page, then click “Download ZIP” this will allow you to access each file individually.

The Linked-List-master folder contains the Linked Folder which contains my source code.

If you would like to access the files without downloading the ZIP, instructions are below.

An Assessment documentation is enclosed in the repository.

To access Assessment documentation which is on Word, click on the file named “Linked List.docx” and also download that file from there.