Nicholas Arnaud
12/14/2016
NA1163
Linked Lists

I. Description of problem

-Students are to be able to work through to create a Linked List to understand how to create program to organize information given into a singular list.

II. Input info

- *Not Applicable

III. Output info

- Outputs information given that was inputted into a list and redisplays information as a list

IV. User Interface

-*Not Applicable

V. System Architecture

V.1 ListIterator.h

```
-#pragma once
#include <iostream>


template<typename Type>
struct nodeType
{
	Type info;
	nodeType<Type> *link;
};
```

```
template<typename Type>
class linkedListIterator
{
private:
        nodeType<Type> * current; //pointer to point to the current node in the
linked list

public:
        //Default constructor
        //Postcondition: current = NULL
        linkedListIterator() {}

        //Constructor with parameter
        //Postcondition: current = node
        linkedListIterator(nodeType<Type> *node)
        {
                current = node;
        }

        //Function to overload the dereferencing operator *
        //Postcondition:Returns the info contained in the node
        Type operator*()
        {
                if (current == NULL)
                        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 == (linkedListIterator<Type>& list) const
        {
                if (list.current == current)
                {
                        return true;
                }
                return false;
        }

        //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 != (linkedListIterator<Type>& list) const
        {
                if (list.current != current)
                {
                        return true;
                }
                return false;
        }
};


template<typename Type>
class linkedListType
{
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

public:
        //Overload the assignment operator
        const linkedListType<Type>& operator = (const linkedListType<Type>&
otherList)
        {
                while (count != NULL)
```

```
	{
		first->info = otherList first->info;
	}
	return first;
}

//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
{
	if (first == NULL)
		return true;
	return false;
}

//Function to output the data contained in each node
//Postcondition: Node
void print() const
{
	nodeType<Type> * current = new nodeType<Type>;

	for (int i = 1; i <= count; i++)
	{
		std::cout << current->info;
	}
}

//Function to return the number of nodes in the list
//Postcondition: The value of count is returned
int length() const
```

```
{
	return count;
}

//Function to delete all the nodes from the list
//Postcondition: first = NULL, last = NULL, count = 0;
void destroyList()
{
	nodeType<Type> * rekt = first;
	while (rekt != NULL)
	{
		nodeType<Type> * out = rekt;
		rekt = rekt->link;
		delete out;
	}
	initializeList();
}

//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 first->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 last->info;
}

//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)
{
	for (int i = 0; i <= count; i++)
	{
		if (nodeInfo == first->info)
			return true;
	}
	return false;
}

//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& nodeInfo)
{
	nodeType<Type> * newNode;
	newNode = new nodeType<Type>;
	if (count == 0)
	{
		newNode->info = nodeInfo;
		newNode->link = NULL;
		newNode->info = nodeInfo;
		newNode->link = NULL;
		count++;
	}
	else
	{
		newNode->link = first;
		first = newNode;
		first->info = nodeInfo;
		count++;
	}
}

//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& nodeInfo)
        {
                if (count == 0)
                {
                        first->info = nodeInfo;
                        first->link = NULL;
                        last->info = nodeInfo;
                        last->link = NULL;
                        count++;
                }
                else
                {
                        nodeType<Type> * newNode;
                        newNode = new nodeType<Type>;
                        last->link = newNode;
                        last = newNode;

                        if (count == 1)
                        {
                                first->link = newNode;
                        }

                        last->info = nodeInfo;
                        last->link = NULL;
                        count++;
                }
        }

        //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)
        {
                for (int i = 0; i <= count; i++)
                {
```

```
		if (nodeInfo == first->info)
		{
			delete first->info;
			count--;
			First->info = nodeInfo->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()
{
	linkedListIterator<Type> tmp = first;
	return tmp;
}

//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()
{
	linkedListIterator<Type> tmp = last;
	return tmp;
}

//Default constructor
//Initializes the list to an empty state
//Postcondition: first = NULL, last = NULL, count = 0;
linkedListType()
{
	first = new nodeType<Type>;
	last = new nodeType<Type>;
	count = 0;
}

//copy constructor
linkedListType(const linkedListType<Type>& otherList)
{
```

```
            this = otherList;

        }

        //deconstructor
        //Deletes all the nodes from the list
        //Postcondition: The list object is destroyed
        ~linkedListType<Type>() {}

    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;
            count = otherlist.count;
            last = otherlist.last;
        }
    };
```

V.2 Source.cpp

```
-#include <iostream>
#include <cassert>
#include <math.h>
#include "ListIterator.h"


int main()
{
    ////////////////////////////////
    //   LINKED LISTS     //
    ///////////////////////////////
    nodeType<int> * Head;
    nodeType<int> a, b, c;

    c.info = 4;
    b.info = 2;
    a.info = 0;
    Head = &c;
```

```
		c.link = &b;
		b.link = &a;
		a.link = NULL;




		linkedListType<int> *K = new  linkedListType<int>();

		K->insertLast(7);
		K->insertFirst(11);
		K->insertFirst(10);
		K->length();
		K->back();
		K->front();
		K->isEmptyList();
		K->search(7);
		K->print();
		K->end();
		K->begin();
		K->destroyList();


		std::fstream file;
		file.open("LinkedListTest.txt", std::ios_base::out);
		if (file.is_open())
		{
			file << "Length of Linked List is: \n"<<K->length()<< "\n\n\n";
			file << "The last variable of the Linked List is: \n" <<
K->back()<<"\n\n\n";
			file << "The first variable of the Linked List is: \n" << K->front() <<
"\n\n\n";
			file << "Is the List empty?  \n" << K->isEmptyList() << "\n\n\n";
			file << "Searching for '7'... it is found? \n" << K->search(7) <<
"\n\n\n";
		}
		file.close();

	K->destroyList();
```

}

VI. Implementation Documents

-After taking the prototype functions and classes, I created the function definitions and turned the classes to include templates.

VII. Read Me

VII.1 Controls

-*Not Applicable

VII.2 How to obtain Application

-

VII.3 How to use Application

-The User runs the .exe file which will create a .txt file and from there, the user will be able to access the newly created text file and is able to view the test cases.