CS5103

DJ Yuhn

Assignment 1

Stack

Stack is a data structure operating on the premise of a Last In First Out (LIFO) method. My implementation of stack utilizes a linked list with the following four methods: getTop, push, pop, and isEmpty. The procedural algorithm for the stack is as follows:

- getTop:
  - Check to see if stack is empty by returning if the top pointer is equivalent to null pointer.
    - If it is empty, throw an exception.
    - If it is not empty, return the data of the node the top pointer points to.
- push:
  - Create a temporary pointer and have it point to a new node.
  - Insert the data into the new node.
  - Check to see if stack is empty by returning if the top pointer is equivalent to null pointer.
    - If it is empty, have the top pointer point to the temporary node.
    - If it is not empty, have the temporary pointer's node's pointer point to the top pointer's node.
      - Assign the top pointer to point to the temporary pointer's node.
- pop:
  - Check to see if stack is empty by returning if the top pointer is equivalent to null pointer.
    - If it is empty throw an exception.
    - If it is not empty, create a temporary pointer and have it point to the top pointer's node.
      - Have the top pointer point to temporary pointer's node's next-node pointer.
      - Initialize a variable and assign it the value of the data in temporary pointer's node.
      - Delete the node temporary pointer is pointing to.
      - Return the data assigned to the variable.
- isEmpty:
  - Check to see if stack is empty by returning if the top pointer is equivalent to null pointer.

Main.cpp Code below for stack:

```
/***************************************************************************
**
* dygw3
* DJ Yuhn
* 10/26/17
* Assignment 1
****************************************************************************
*/

#include <iostream>
#include "StackLinkList.h"

void tryPop(StackLinkList<int>& list);
void tryTop(StackLinkList<int>& list);

int main() {

        // Testing the implementation of Stack using Linked List
        StackLinkList<int> theStack;

        theStack.push(1);
        theStack.push(2);

        tryPop(theStack);
        tryTop(theStack);

        tryPop(theStack);
        tryPop(theStack);

        system("Pause");
        return 0;
}

// Try to pop the top item from the Stack.
void tryPop(StackLinkList<int>& list)
{
        try {
                std::cout << list.pop() << " was removed from the stack.\n";
        }
        catch (std::out_of_range ex) {
                std::cout << ex.what() << std::endl;
        }
}

// Try to view the top item of the stack.
void tryTop(StackLinkList<int>& list)
{
        try {
                std::cout << list.getTop() << " is at the top of the
Stack.\n";
        }
        catch (std::out_of_range ex) {
                std::cout << ex.what() << std::endl;
        }
}
```

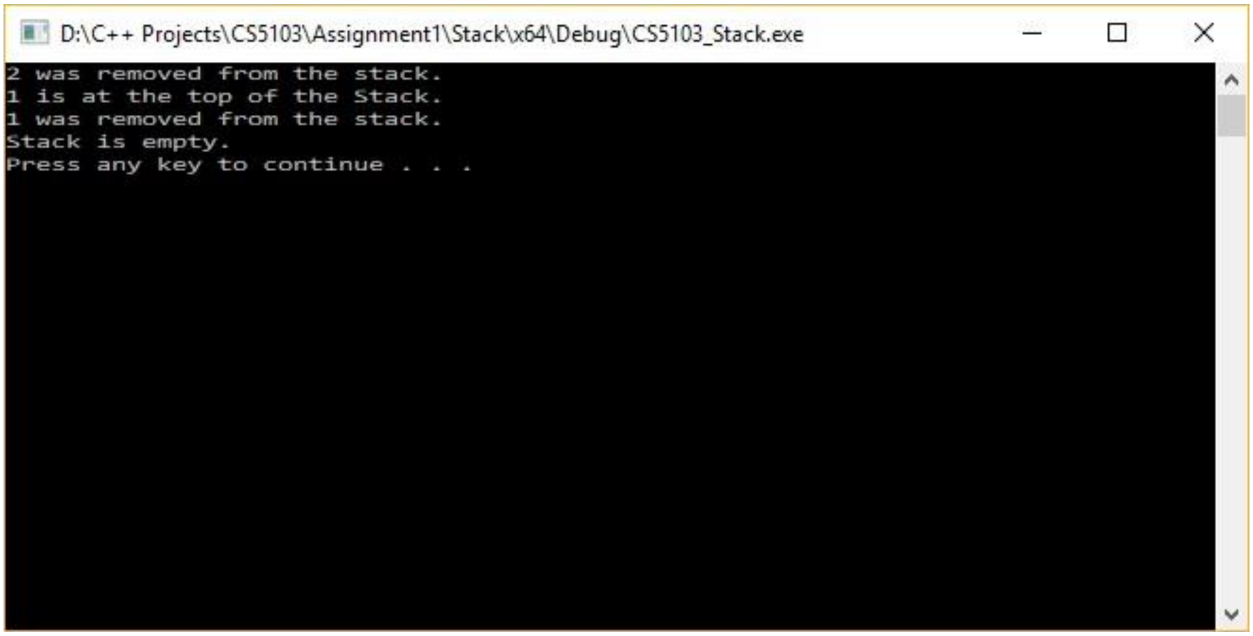
StackLinkList.h Code below for stack:

```
/*************************************************************************
**
* dygw3
* DJ Yuhn
* 10/26/17
* Assignment 1
*************************************************************************
*/

#ifndef STACKLINKLIST_H
#define STACKLINKLIST_H

template<class T>
class StackLinkList {

public:
        StackLinkList() { top = nullptr; }
        ~StackLinkList() { deleteList(); }

        StackLinkList(const StackLinkList &obj);

        StackLinkList& operator=(const StackLinkList& rhs);

        // Get the top node in the stack's data value
        T getTop() const;

        // Add a new node to the top of the stack with a data value
        void push(T value);

        // Remove the node from the top of the stack
        T pop();

        // Check if the linked list is empty
        bool isEmpty() const { return top == nullptr; }

private:
        // Structure for nodes in the list to hold data
        struct Node {
                T data;
                Node *next = nullptr;
        };

        Node *top; // Pointer to top of the stack

        // Delete entirety of the list
        void deleteList();

        // Deep copy the linked lists
        void deepCopy(const StackLinkList& obj);

};

template<class T>
StackLinkList<T>::StackLinkList(const StackLinkList &obj) {
        if (obj.top == nullptr)
```

```
                this->top = nullptr;

        // Perform deep copy
        else
                this->deepCopy(obj)
};

template<class T>
StackLinkList<T>& StackLinkList<T>::operator=(const StackLinkList& rhs) {
        if (this != &rhs) {
                this->deleteList();
                this->deepCopy(rhs);
        }

        return *this;
};

template<class T>
T StackLinkList<T>::getTop() const {
        // Check to see if stack is empty and throw exception if true.
        if (isEmpty())
                throw std::out_of_range("Stack is empty.");

        else
                return top->data;
};

template<class T>
void StackLinkList<T>::push(T value) {
        // Create new node and insert the value into data
        Node *temp = new Node;
        temp->data = value;

        // If stack is empty point top to temp
        if (isEmpty())
                top = temp;

        // Have temp next pointer point to top. Aassign top to temp.
        else {
                temp->next = top;
                top = temp;
        }
};

template<class T>
T StackLinkList<T>::pop() {
        // Check to see if stack is empty and throw exception if true
        if (isEmpty())
                throw std::out_of_range("Stack is empty.");

        // If stack is not empty, delete the top node and return its value
        else {
                Node *temp = top;
                top = temp->next;

                T data = temp->data;
```

```
                delete temp;
                temp = nullptr;

                return data;
        }
}

template<class T>
void StackLinkList<T>::deleteList() {
        Node *temp = nullptr;

        // Check if list has any nodes and continue to delete until empty
        while (!isEmpty()) {
                temp = top;
                top = temp->next;

                delete temp;
        }

        top = nullptr;
        temp = nullptr;
}

template<class T>
void StackLinkList<T>::deepCopy(const StackLinkList& obj) {
        Node* p1; // Pointer for this current node
        Node* o1; // Pointer for obj next node

        this->top = new Node;
        this->top->data = obj.top->data;

        p1 = top;
        o1 = obj.top->next;

        while (o1 != nullptr) {
                p1->next = new Node;
                p1->next->data = o1->data;
                p1 = p1->next;
                o1 = o1->next;
        }
}
;

#endif
```

Output for my stack implementation:



Queue

Queue is a data structure operating on the premise of a First In First Out (FIFO) method. My implementation of queue utilizes a linked list with the following four methods: getFront, push, pop, and isEmpty. The procedural algorithm for the queue is as follows:

- getFront:
    - Check to see if queue is empty by returning if the top pointer is equivalent to null pointer.
        - If it is empty, throw an exception.
        - If it is not empty, return the data of the node the front pointer points to.
- push:
    - Create a temporary pointer and have it point to a new node.
    - Insert the data into the new node.
    - Check to see if queue is empty by returning if the front pointer is equivalent to null pointer.
        - If it is empty, have the front pointer and the back pointer point to the temporary node.
        - If it is not empty, have the back pointer's node's next pointer point to the temporary pointer's node.
            - Assign the back pointer to point to the temporary pointer's node.
- pop:
    - Check to see if queue is empty by returning if the front pointer is equivalent to null pointer.
        - If it is empty throw an exception.
        - If it is not empty, create a temporary pointer and have it point to the front pointer's node.
            - Have the front pointer point to temporary pointer's node's next-node pointer.
            - Initialize a variable and assign it the value of the data in temporary pointer's node.
            - Delete the node temporary pointer is pointing to.
            - If the front pointer is now a null pointer, assign the back pointer as a null pointer.
            - Return the data assigned to the variable.
- isEmpty:
    - Check to see if stack is empty by returning if the top pointer is equivalent to null pointer.

Main.cpp Code below for queue:

```
/***************************************************************************
**
* dygw3
* DJ Yuhn
* Assignment 1
****************************************************************************
*/

#include <iostream>
#include "QueueLinkList.h"

void tryPop(QueueLinkList<int>& list);
void tryFront(QueueLinkList<int>& list);

int main() {

        // Testing the implementation of Stack using Linked List
        QueueLinkList<int> theQueue;

        theQueue.push(1);
        theQueue.push(2);

        tryPop(theQueue);
        tryFront(theQueue);

        tryFront(theQueue);
        tryPop(theQueue);

        tryPop(theQueue);

        system("Pause");
        return 0;
}

// Try to pop the top item from the Stack.
void tryPop(QueueLinkList<int>& list)
{
        try {
                std::cout << list.pop() << " was removed from the queue.\n";
        }
        catch (std::out_of_range ex) {
                std::cout << ex.what() << std::endl;
        }
}

// Try to view the top item of the stack.
void tryFront(QueueLinkList<int>& list)
{
        try {
                std::cout << list.getFront() << " is at the front of the
queue.\n";
        }
        catch (std::out_of_range ex) {
                std::cout << ex.what() << std::endl;
        }
```

```
}
```

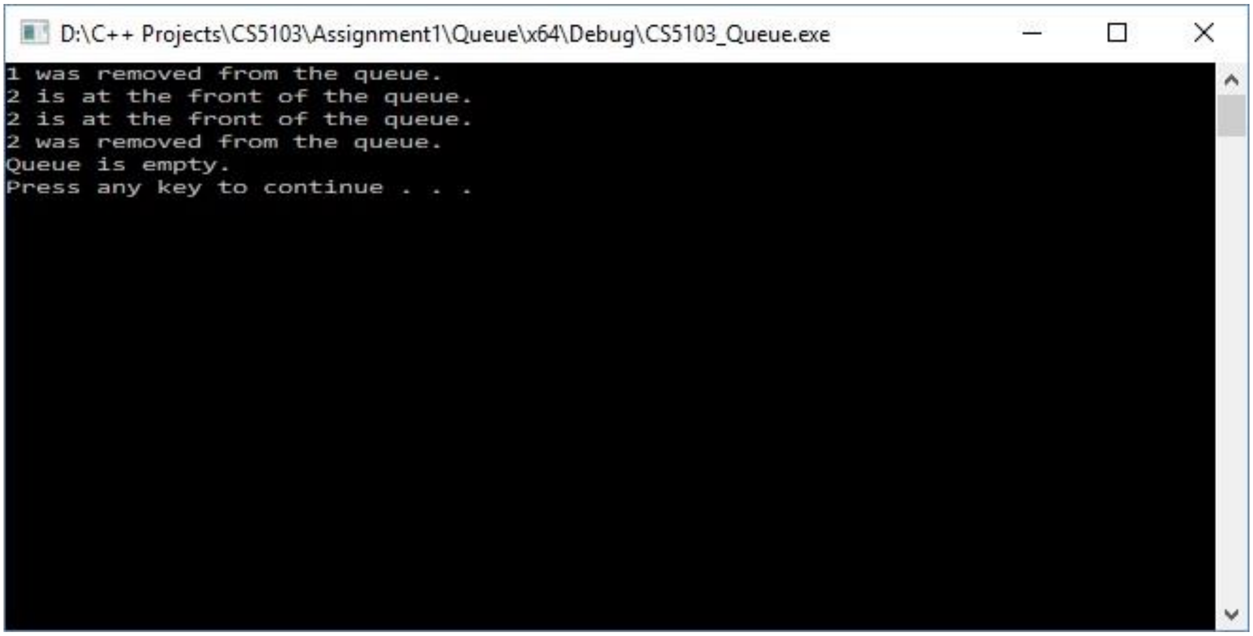
QueueLinkList.h Code below for queue:

```
/***************************************************************************
**
* dygw3
* DJ Yuhn
* Assignment 1
***************************************************************************
*/

#ifndef QUEUELINKLIST_H
#define QUEUELINKLIST_H


template<class T>
class QueueLinkList {

public:
        QueueLinkList() { front = nullptr, back = nullptr; }
        ~QueueLinkList() { deleteList(); }

        QueueLinkList(const QueueLinkList &obj);

        QueueLinkList& operator=(const QueueLinkList& rhs);

        // Get the front node in the queue's data value
        T getFront() const;

        // Add a new node to the end of the queue with a data value
        void push(T value);

        // Remove the node from the front of the queue
        T pop();

        // Check if the linked list is empty
        bool isEmpty() const { return front == nullptr; }

private:
        // Structure for nodes in the list to hold data
        struct Node {
                T data;
                Node *next = nullptr;
        };

        Node *front; // Pointer to front of the queue
        Node *back; // Pointer to back of queue

        // Delete entirety of the list
        void deleteList();

        // Deep copy the linked lists
        void deepCopy(const QueueLinkList& obj);

};

template<class T>
QueueLinkList<T>::QueueLinkList(const QueueLinkList &obj) {
```

```
	if (obj.front != nullptr)
		this->deepCopy(obj);
};

template<class T>
QueueLinkList<T>& QueueLinkList<T>::operator=(const QueueLinkList& rhs) {
	if (this != &rhs) {
		this->deleteList();
		this->deepCopy(rhs);
	}

	return *this;
};

template<class T>
T QueueLinkList<T>::getFront() const {
	// Check to see if queue is empty and throw exception if true.
	if (isEmpty())
		throw std::out_of_range("Queue is empty.");

	else
		return front->data;
};

template<class T>
void QueueLinkList<T>::push(T value) {
	// Create new node and insert the value into data
	Node *temp = new Node;
	temp->data = value;

	// If queue is empty point front and back to temp
	if (isEmpty()) {
		front = temp;
		back = temp;
	}

	// Have back next pointer point to temp. Assign back to temp.
	else {
		back->next = temp;
		back = temp;
	}
};

template<class T>
T QueueLinkList<T>::pop() {
	// Check to see if queue is empty and throw exception if true
	if (isEmpty())
		throw std::out_of_range("Queue is empty.");

	// If queue is not empty, delete the front node and return its value
	else {
		Node *temp = front;
		front = temp->next;

		T data = temp->data;

		delete temp;
```

```
		temp = nullptr;

		// If front is now a nullptr, insure back is nullptr
		if (front == nullptr)
			back = nullptr;

		return data;
	}
};

template<class T>
void QueueLinkList<T>::deleteList() {
	Node *temp = nullptr;

	// Check if list has any nodes and continue to delete until empty
	while (!isEmpty()) {
		temp = front;
		front = temp->next;

		delete temp;
	}

	back = nullptr;
	temp = nullptr;
};

template<class T>
void QueueLinkList<T>::deepCopy(const QueueLinkList& obj) {
	Node* p1; // Pointer for this current node
	Node* o1; // Pointer for obj next node

	this->front = new Node;
	this->front->data = obj.front->data;

	p1 = front;
	o1 = obj.front->next;

	while (o1 != nullptr) {
		p1->next = new Node;
		p1->next->data = o1->data;

		this->back = p1;
		p1 = p1->next;
		o1 = o1->next;
	}
};


#endif
```

Output for my queue implementation:



Doubly Linked List

Doubly linked list is a data structure operating on the premise of nodes connecting together with previous and next pointers. My implementation of the doubly linked list used the following seven methods: pushFront, pushBack, insertAt, deleteFront, deleteBack, deleteAt, isEmpty. There is a private variable that keeps count of the nodes. The procedural algorithm for the doubly linked list is as follows:

- pushFront:
  - Create a temporary pointer and have it point to a new node.
  - Insert the data into the new node.
  - Check to see if list is empty by returning if the front pointer is equivalent to null pointer.
    - If it is empty, have the front and back pointers point to the temporary node.
    - If it is not empty, have the temporary pointer's node's next pointer point to front.
      - Have front pointer's node's previous pointer point to temp.
      - Assign the front pointer to point to the temporary pointer's node.
  - Increment the number of nodes by 1.
- pushBack:
  - Create a temporary pointer and have it point to a new node.
  - Insert the data into the new node.
  - Check to see if list is empty by returning if the front pointer is equivalent to null pointer.
    - If it is empty, have the front and back pointers point to the temporary node.
    - If it is not empty, have the back pointer's node's next pointer point to the temporary pointer's node.
      - Have temporary pointer's node's previous pointer point to back.
      - Assign the back pointer to point to the temporary pointer's node.
  - Increment the number of nodes by 1.
- insertAt:
  - If position entered is less than 0.
    - Throw an exception.
  - If position entered is equivalent to 0.
    - Call pushFront method.
  - If position entered is less than or equal to the number of nodes in the list.
    - Create the before pointer and have it point to front pointer's node.
    - Create the after pointer and have it be a null pointer.
    - Create the newNode pointer and have it pointer to a new node.
    - Insert the data into the new node.
    - Traverse the list with a for loop, starting at 0 and increment the variable by one with the end condition being when the variable is less than the position entered minus one.
      - For each loop, assign the before pointer to before pointer's node's next pointer.
    - If before pointer is not equivalent to null pointer.

            - Have after pointer point to before pointer's node's next pointer.
            - Have newNode pointer's node's previous pointer point to before pointer.
            - Have newNode pointer's node's next pointer point to after pointer.
            - Have before pointer's node's next pointer point to newNode.
            - If after pointer is not a null pointer.
                - Have after pointer's node's previous pointer point to newNode.
            - If after pointer is a null pointer.
                - Have back pointer point to newNode.
            - Increment number of nodes by 1.
        - If before pointer is a null pointer.
            - Have front pointer point to newNode.
            - Have back pointer point to newNode.
            - Increment number of nodes by 1.
    - If position is greater than the number of nodes in the list.
        - Throw an exception.
- deleteFront:
    - Check to see if list is empty by returning if the front pointer is equivalent to null pointer.
        - If it is empty.
            - Throw an exception.
        - If it is not empty.
            - Create a temporary pointer and have it point to the front pointer's node.
            - Have the front pointer point to temporary pointer's node's next pointer.
            - If front pointer is a null pointer.
                - Assign back pointer to be a null pointer.
            - If front pointer is not a null pointer.
                - Have front pointer's node's previous pointer be a null pointer.
            - Create a variable and assign it temp pointer's node's data.
            - Delete temporary pointer's node.
                - Assign temporary pointer to be a null pointer.
            - Decrement number of nodes by 1.
            - Return the variable with the data.
- deleteBack:
    - Check to see if list is empty by returning if the front pointer is equivalent to null pointer.
        - If it is empty.
            - Throw an exception.
        - If it is not empty.
            - Create a temporary pointer and have it point to the back pointer's node.
            - Have the back pointer point to temporary pointer's node's previous pointer.
            - If back pointer is a null pointer.
                - Assign front pointer to be a null pointer.

            - If back pointer is not a null pointer.
                - Have back pointer's node's next pointer be a null pointer.
            - Create a variable and assign it temp pointer's node's data.
            - Delete temporary pointer's node.
                - Assign temporary pointer to be a null pointer.
            - Decrement number of nodes by 1.
            - Return the variable with the data.
- deleteAt:
    - Check to see if list is empty by returning if the front pointer is equivalent to null pointer.
        - If it is empty.
            - Throw an exception.
    - If position entered is equivalent to 0.
        - Call deleteFront method.
    - If list is not empty.
        - Create search pointer and have it point to front.
        - Create temp1 pointer and have it be a null pointer.
        - Create temp2 pointer and have it be a null pointer.
        - If position is less than or equal to the number of nodes in the list and position is greater than 0.
            - Traverse the list with a for loop, starting at 0 and increment the variable by one with the end condition being when the variable is less than the position entered.
                - For each loop assign search pointer to search pointer's node's next pointer.
            - Create a variable to hold search pointer's node's data.
            - Have temp1 pointer point to search pointer's node's previous pointer.
            - Have temp2 pointer point to search pointer's node's next pointer.
            - If temp1 and temp2 pointers are not null pointers.
                - Have temp1 pointer's node's next pointer point to temp2 pointer.
                - Have temp2 pointer's node's previous pointer point to temp1 pointer.
            - If temp1 is not a null pointer and temp2 is a null pointer.
                - Have temp1 pointer's node's next pointer be a null pointer.
            - If temp1 is a null pointer and temp2 is not a null pointer.
                - Have temp2 pointer's node's previous pointer be a null pointer.
            - Delete the search pointer's node.
            - Decrement the number of nodes by 1.
            - Return the variable holding the data.
        - If position is less than 0 or greater than the number of nodes in the list.
            - Throw an exception.

Main.cpp code for doubly linked list below:

```
/***********************************************************************
**
* dygw3
* DJ Yuhn
* Assignment 1
************************************************************************
*/

#include <iostream>
#include "DoublyLinkedList.h"


int main() {

        DoublyLinkedList<int> dblList;

        std::cout << dblList.getSize() << std::endl;

        dblList.pushFront(1);
        dblList.pushBack(3);
        dblList.insertAt(2, 1);
        dblList.insertAt(0, 0);

        std::cout << dblList.getSize() << std::endl;

        std::cout << dblList.deleteFront() << " was deleted from the front."
<< std::endl;
        std::cout << dblList.deleteAt(1) << " was deleted from index 1." <<
std::endl;
        std::cout << dblList.deleteBack() << " was deleted from the back." <<
std::endl;

        system("Pause");
        return 0;
}
```

DoublyLinkedList.h code for doubly linked list below:

```
/*************************************************************************
**
* dygw3
* DJ Yuhn
* Assignment 1
**************************************************************************
*/

#ifndef DOUBLYLINKEDLIST_H
#define DOUBLYLINKEDLIST_H


template<class T>
class DoublyLinkedList {

public:
        DoublyLinkedList() { front = nullptr, back = nullptr, numNode = 0; }
        ~DoublyLinkedList() { deleteList(); }

        DoublyLinkedList(const DoublyLinkedList &obj);

        DoublyLinkedList& operator=(const DoublyLinkedList& rhs);

        // Returns number of nodes in the list
        int getSize() const { return numNode; }

        // Add new nodes
        void pushFront(T value);
        void pushBack(T value);
        void insertAt(T value, int pos);

        // Remove nodes
        T deleteFront();
        T deleteBack();
        T deleteAt(int pos);

        // Check if the linked list is empty
        bool isEmpty() const { return front == nullptr; }

private:
        // Structure for nodes in the list to hold data
        struct Node {
                T data;

                Node *next = nullptr;
                Node *prev = nullptr;
        };

        Node *front; // Pointer to front of the list
        Node *back; // Pointer to back of list
        int numNode;

        // Delete entirety of the list
        void deleteList();
```

```
        // Deep copy the linked lists
        void deepCopy(const DoublyLinkedList& obj);

};

template<class T>
DoublyLinkedList<T>::DoublyLinkedList(const DoublyLinkedList &obj) {
        if (obj.front != nullptr)
                this->deepCopy(obj);
};

template<class T>
DoublyLinkedList<T>& DoublyLinkedList<T>::operator=(const DoublyLinkedList&
rhs) {
        if (this != &rhs) {
                this->deleteList();
                this->deepCopy(rhs);
        }

        return *this;
};

// Adds data to the head of the list
template<class T>
void DoublyLinkedList<T>::pushFront(T value){
        // Create new node and insert the value into data
        Node *temp = new Node;
        temp->data = value;

        // If list is empty point front and back to temp
        if (isEmpty()) {
                front = temp;
                back = temp;
        }

        // Have temp next pointer point to front, front prev point to temp
        else {
                temp->next = front;
                front->prev = temp;
                front = temp;
        }

        // Increment number of nodes
        numNode++;
};

// Adds data to the tail of the list
template<class T>
void DoublyLinkedList<T>::pushBack(T value) {
        // Create new node and insert the value into data
        Node *temp = new Node;
        temp->data = value;

        // If list is empty point front and back to temp
        if (isEmpty()) {
                front = temp;
                back = temp;
```

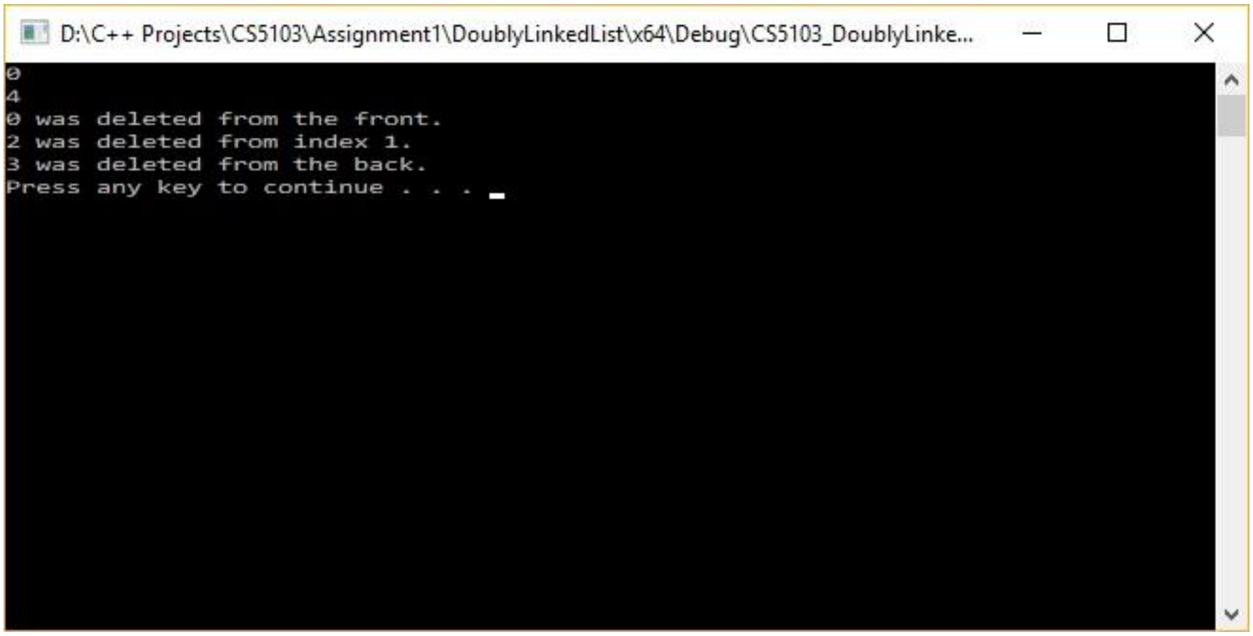
```
	}

	// Have back next pointer point to temp, temp prev point to back.
	else {
		back->next = temp;
		temp->prev = back;
		back = temp;
	}

	// Increment number of nodes
	numNode++;
};

// Inserts data at the position. (Indexing begins at 0)
template<class T>
void DoublyLinkedList<T>::insertAt(T value, int pos) {
	// Insure valid number. Must be positive
	if (pos < 0)
		throw std::out_of_range("Negative integer entered.");

	// If index is 0, push to front
	else if (pos == 0)
		pushFront(value);

	else if (pos <= getSize()) {
		Node *before = front;
		Node *after = nullptr;
		Node *newNode = new Node;
		newNode->data = value;

		// Traverse list until node before desired position.
		for (int i = 0; i < (pos - 1); i++)
			before = before->next;

		if (before != nullptr) {
			after = before->next; // Hold the next node
			newNode->prev = before; // Assign newNode prev pointer
to the before
			newNode->next = after; // Assign newNode next pointer
to after
			before->next = newNode; // Assign before next pointer
to the new node

			if (after != nullptr)
				after->prev = newNode; // Assign after previous
pointer to the new node
			else
				back = newNode; // End of the list

			numNode++;
		}

		else {
			front = newNode;
			back = newNode;
			numNode++;
		}
```

```
        }

        else
                throw std::out_of_range("Inserting outside range of the
list.");
};

// Deletes data at front of list and returns the value
template<class T>
T DoublyLinkedList<T>::deleteFront() {
        // Check to see if list is empty and throw exception if true
        if (isEmpty())
                throw std::out_of_range("List is empty.");

        // If list is not empty, delete the front node and return its value
        else {
                Node *temp = front;
                front = temp->next;

                // If front is now a nullptr, insure back is nullptr
                if (front == nullptr)
                        back = nullptr;
                else
                        front->prev = nullptr;

                T data = temp->data;

                delete temp;
                temp = nullptr;

                numNode--; // Decrement number of nodes

                return data;
        }
};

// Deletes data at back of list and returns the value
template<class T>
T DoublyLinkedList<T>::deleteBack() {
        // Check to see if list is empty and throw exception if true
        if (isEmpty())
                throw std::out_of_range("List is empty.");

        // If list is not empty, delete the back node and return its value
        else {
                Node *temp = back;
                back = temp->prev;

                // If back is now a nullptr, insure front is nullptr
                if (back == nullptr)
                        front = nullptr;
                else
                        back->next = nullptr;

                T data = temp->data;

                delete temp;
```

```
		temp = nullptr;

		numNode--; // Decrement number of nodes

		return data;
	}
};

// Deletes data at the specified position and returns the value
template<class T>
T DoublyLinkedList<T>::deleteAt(int pos)
{
	// Check to see if list is empty and throw exception if true
	if (isEmpty())
		throw std::out_of_range("List is empty.");

	// If position is 0 delete front
	else if (pos == 0)
		return deleteFront();

	// If list is not empty attempt search
	else {
		Node *search = front;
		Node *temp1 = nullptr;
		Node *temp2 = nullptr;

		// If position is less than or equal to total number of nodes,
search
		if (pos <= getSize() && pos > 0) {
			for (int i = 0; i < pos; i++)
				search = search->next;

			T data = search->data;

			temp1 = search->prev;
			temp2 = search->next;

			if (temp1 != nullptr && temp2 != nullptr) {
				temp1->next = temp2;
				temp2->prev = temp1;
			}
			else if (temp1 != nullptr && temp2 == nullptr) {
				temp1->next = nullptr;
			}
			else if (temp1 == nullptr && temp2 != nullptr) {
				temp2->prev = nullptr;
			}

			delete search;
			numNode--;

			return data;

		}

		else
```

```
                        throw std::out_of_range("Deleting outside range of the
list");
        }
};

template<class T>
void DoublyLinkedList<T>::deleteList() {
        Node *temp = nullptr;

        // Check if list has any nodes and continue to delete until empty
        while (!isEmpty()) {
                temp = front;
                front = temp->next;

                delete temp;
        }

        back = nullptr;
        temp = nullptr;
}

template<class T>
void DoublyLinkedList<T>::deepCopy(const DoublyLinkedList& obj) {
        Node* p1; // Pointer for this current node
        Node* o1; // Pointer for obj next node

        this->front = new Node;
        this->front->data = obj.front->data;

        p1 = front;
        o1 = obj.front->next;

        while (o1 != nullptr) {
                p1->next = new Node;
                p1->next->data = o1->data;

                this->back = p1;
                p1 = p1->next;
                o1 = o1->next;
        }
};


#endif
```

Output for my doubly linked list implementation:



References

https://codereview.stackexchange.com/questions/26451/copy-constructor-and-assignment-operator-for-doubly-linked-list
http://www.cplusplus.com/forum/general/13222/
https://stackoverflow.com/questions/7811893/creating-a-copy-constructor-for-a-linked-list
https://stackoverflow.com/questions/34963158/doubly-linked-list-template-copy-constructor-assignment-operator
https://codereview.stackexchange.com/questions/136077/insert-a-node-at-the-tail-of-a-linked-list