**Program Design**

Garrett Wells

CS121

3-4-19

***Bank Simulation Application File:***

This takes an input bank file of the format:

Entry Time    Name    Activity    Transactions

The program then uses the data in the file to calculate how long the customers in the bank are in the bank and in what order they finish their transactions.

|  |  |  |
| --- | --- | --- |
| Return Type | Name | Function Description |
| int | main(void) | Takes in an input file of customers, runs a simulation using the data, then prints the results |
| void | printOutput() | Prints the results of the simulation calculations |
| bool | linesEmpty(Queue line1, Queue line2, Queue line3) | Returns true if all the lines are empty |
| void | printCustomer(Node customer, int exitTime) | Prints the information for a customer leaving the bank, including exit time |
| void | AddToLine(Node customer) | Adds a customer to the end of the shortest line |
| void | PrintLines() | Prints the customers currently in the bank and what line they are in |
| void | runTests() | Prints test results of LinkedList and Queue |

***Tester Class:***

Tests the implementation of Queue and LinkedList classes.

|  |  |  |
| --- | --- | --- |
| *Return Type* | *Name* | *Function Description* |
| *(Constructor)* | Tester() | Runs and prints the results from testing the queue and linked list implementation |
| int | testLinkedList() | Tests the various add, remove, and print functions of LinkedList |
| int | testQueue() | Tests Enqueue() and Dequeue() functions of Queue class |

***LinkedList Class:***

|  |  |  |
| --- | --- | --- |
| Return Type | Name | Function Description |
| (Constructor) | LinkedList() | Creates a new, empty Linked List object. |
| void | AddToEnd(string name, char line, int num) | Creates a new Node and adds it to the end of the Linked List |
| void | AddToFront(string name, char line, int num) | Creates a new Node and adds it to the front of the Linked List |
| Node | RemoveFromFront() | Removes a node from the front of the Linked List and returns the name of the customer |
| Node | RemoveFromBack() | Removes a node from the back of the Linked List and returns the name of the customer |
| Node | Peek() | Returns a pointer to the Node at the front of the line |
| int | getSize() | Returns how many Nodes are in the list |
| void | Print() | Prints the Nodes in the list |

***Queue Class:***

|  |  |  |
| --- | --- | --- |
| Return Type | Name | Function Description |
| (Constructor) | Queue() | Creates a new, empty Queue object |
| void | Enqueue(string name, char line, int num) | Adds a new customer to the end of the Queue |
| Node | Remove() | Remove a customer from the end of the line and return their name |
| Node | Dequeue() | Remove a customer from the front of the Queue and return their name |
| int | getSize() | Returns the number of customers in line |
| void | Print() | Prints the contents of the Queue |

**Programming Log:**

2-27-19

1hr – Created program design and implemented LinkedList and Queue classes

2-28-19

2hr – Refined program design and rewrote LinkedList and Queue implementation. Also wrote code for Tester Class and debugged LinkedList and Queue.

3-1-19

0.75hr – Began implementing Bank Simulation.

3-2-19

4hrs – Finished debugging logical errors and formatting program output

Estimated Time to Complete = 6hrs

**Total Required Time = 8hrs**

*/\*  
  
 Application File for Bank  
 @author: Garrett Wells  
 @date: 3-4-19  
  
\*/*#include **"queue.cpp"**#include **"tester.cpp"**#include **<fstream>  
  
bool** linesEmpty(Queue line1, Queue line2, Queue line3);  
**void** printCustomer(Node customer, **int** exitTime);  
**void** AddToLine(Node nextUp);  
**void** runTests(**void**);  
**void** PrintLines(**void**);  
  
Queue expressLine;  
Queue line2;  
Queue line3;  
Queue waiting;  
  
**int** main(**void**){  
 *// Run Tests  
 //runTests();  
  
 //Run Bank Simulation  
 // temporary variables to hold data* **int** entryTime;  
 string customerName;  
 **char** transactionType;  
 **int** numTrans;  
  
 *// Read from file --------------------------------* ifstream input(**"/Users/garrettwells/Documents/2018-19\_UofI/2nd\_Semester/CS121/Projects/Project#3/bank1.dat"**);  
 **while**(!input.eof() && input.is\_open()){  
 **for**(**int** i = 0; i < 4; i++){  
 **if**(i == 0)  
 input >> entryTime;  
 **if**(i == 1)  
 input >> customerName;  
 **if**(i == 2)  
 input >> transactionType;  
 **if**(i == 3)  
 input >> numTrans;  
  
 }  
 waiting.Enqueue(entryTime, customerName, transactionType, numTrans);  
 }  
  
 input.close();  
  
  
 *// Banking Loop -----------------------------------* **int** clk = 0;  
 **int** expressLineStart = 0;  
 **int** line2Start = 0;  
 **int** line3Start = 0;  
  
 **while**((waiting.getSize() > 0) || !linesEmpty(expressLine, line2, line3)){  
  
 **bool** someLineEmpty = expressLine.getSize() == 0 || line2.getSize() == 0

|| line3.getSize() == 0;  
 **bool** someLineExcess = (expressLine.getSize() > 1) || (line2.getSize() > 1)

|| (line3.getSize() > 1);  
  
 *// If there is an empty line, and there is some line with more than 1 node, then transfer a node from the longest line to the shortest* **while**(someLineEmpty && someLineExcess){  
  
 **bool** line2IsLongest = (line2.getSize() > line3.getSize())

&& (line3.getSize() > expressLine.getSize());  
 **bool** line3IsLongest = ((line3.getSize() > line2.getSize())

&& (line3.getSize() > expressLine.getSize()));  
 **bool** expressIsLongest = !line2IsLongest && !line3IsLongest;  
  
 **bool** line2Shortest = (line2.getSize() < line3.getSize())

&& (line2.getSize() < expressLine.getSize());  
 **bool** line3Shortest = (line3.getSize() < line2.getSize())

&& (line3.getSize() < expressLine.getSize());  
 **bool** expressShortest = !line2Shortest && !line3Shortest;  
  
 *// Get the node to transfer to the empty line  
 // Check if shortest line is express, if it is we need to put some restrictions on the transfer* **if**(expressShortest) {  
 *// Add the node to the empty line* Node temp;  
 **if**(line2IsLongest){  
 temp = line2.Remove();  
  
 }**else if**(line3IsLongest){  
 temp = line3.Remove();  
  
 }**else** {  
 temp = expressLine.Remove();  
  
 }  
  
 **bool** validTrans = ((temp.transactionType == **'C'**)

|| (temp.transactionType == **'D'**));  
  
 **if**(validTrans){  
 *// Add the node* AddToLine(temp);  
  
 }**else**{ *// Look for another possible transfer  
 /\*If second longest line has more than 2 nodes and the node has valid transaction - > transfer it\*/  
 // Move from second longest to express line* **if**(line2IsLongest && (line3.getSize() > 1)){  
 validTrans = (line3.Peek().transactionType == **'C'**)

|| (line3.Peek().transactionType == **'D'**);  
 **if**(validTrans) AddToLine(line3.Remove());  
  
 }**else if**(line2.getSize() > 1){

*// Try line 2* validTrans = (line2.Peek().transactionType == **'C'**)

|| (line2.Peek().transactionType == **'D'**);  
 **if**(validTrans) AddToLine(line2.Remove());  
  
 }**else**{  
 *// No alternative transfer* **break**;  
 }  
 }  
  
 }**else**{  
 *// Add the node to whatever the shortest line is* Node temp;  
 **if**(line2IsLongest){  
 temp = line2.Remove();  
  
 }**else if**(line3IsLongest){  
 temp = line3.Remove();  
  
 }**else** {  
 temp = expressLine.Remove();  
  
 }  
 AddToLine(temp);  
 }  
  
  
 someLineEmpty = expressLine.getSize() == 0 || line2.getSize() == 0

|| line3.getSize() == 0;  
 someLineExcess = (expressLine.getSize() > 1) || (line2.getSize() > 1)

|| (line3.getSize() > 1);  
  
 }  
  
 *// Add new customer to line* **if**(waiting.getSize() > 0){  
 Node first = waiting.Peek();  
  
 **if**(first.entryTime == clk) {  
  
 **while** (first.entryTime == clk) {  
 cout << **"-------------------------------------"** << **"\n"** << **"| Adding New Customer to Line: "** << waiting.Peek().name

<< **" |\n"** << endl  
 << **"-------------------------------------"** << endl << endl;  
  
 AddToLine(waiting.Dequeue());  
  
  
 **if**(waiting.getSize() > 0) {  
 first = waiting.Peek();  
 }**else break**;  
 }  
 }  
  
 }

*// Check for the customers who have completed their transactions  
 // Remove the customers who are done  
 // Print their information* **if**(linesEmpty(expressLine, line2, line3)){clk++; **continue**;}  
  
 **if**(expressLine.getSize() > 0){  
 Node expressLineFront = expressLine.Peek();  
  
 **bool** transactionsCompleted = (expressLineFront.entryTime

+ (expressLineFront.transactions \* 2) == clk);  
  
 **if**(transactionsCompleted) {  
 expressLineFront = expressLine.Dequeue();  
 printCustomer(expressLineFront, clk);  
 }  
 }  
  
 **if**(line2.getSize() > 0){  
 Node line2Front = line2.Peek();  
  
 **bool** transactionsCompleted = (line2Front.entryTime

+ (line2Front.transactions \* 4) == clk);  
  
 **if**(transactionsCompleted){  
 line2Front = line2.Dequeue();  
 printCustomer(line2Front, clk);  
 }  
 }  
  
 **if**(line3.getSize() > 0){  
 Node line3Front = line3.Peek();  
  
 **bool** transactionsCompleted = (line3Front.entryTime

+ (line3Front.transactions \* 4) == clk);  
  
 **if**(transactionsCompleted){  
 line3Front = line3.Dequeue();  
 printCustomer(line3Front, clk);  
 }  
 }  
  
 clk+=1;  
 }  
}  
  
*// Run Tests***void** runTests(**void**){  
 Tester a;  
}  
  
*// Check if lines are empty and returns true if all customers have finished their transactions***bool** linesEmpty(Queue line1, Queue line2, Queue line3){  
 **return** ((line1.getSize() == 0) && (line2.getSize() == 0) && (line3.getSize() == 0));  
}  
  
*// Print data stored by the customer***void** printCustomer(Node customer, **int** exitTime){  
 cout << **" |---------- Customer Leaving ----------"** << endl  
 << **" |Name: "** << customer.name << endl  
 << **" |Entry Time: "** << customer.entryTime << endl  
 << **" |Transaction Type: "** << customer.transactionType << endl  
 << **" |Number of Transactions: "** << customer.transactions << endl  
 << **" |Exit Time: "** << exitTime << endl  
 << **" |--------------------------------------"** << endl << endl;  
}  
  
*// Print Lines***void** PrintLines(**void**){  
 cout << **"--------- Express Line: -----------"** << endl;  
 expressLine.Print();  
  
 cout << **"----------- Line 2: ----------"** << endl;  
 line2.Print();  
  
 cout << **"----------- Line 3: ----------"** << endl;  
 line3.Print();  
  
}  
  
*// Find shortest line to add Node to***void** AddToLine(Node nextUp){  
  
 **int** expressLength = expressLine.getSize();  
 **int** line2Length = line2.getSize();  
 **int** line3Length = line3.getSize();  
  
 **bool** expressShortest = (expressLength <= line2Length)

&& (expressLength <= line3Length);  
 **bool** expressEligible = (nextUp.transactionType == **'C'**

|| nextUp.transactionType == **'D'**);  
  
 **if**(expressEligible && expressShortest) {  
 *// Add node to expressLine* expressLine.Enqueue(nextUp.entryTime, nextUp.name, nextUp.transactionType, nextUp.transactions);  
  
 }**else if**((line2Length < line3Length)){  
 *// Add node to line2* line2.Enqueue(nextUp.entryTime, nextUp.name, nextUp.transactionType, nextUp.transactions);  
  
 }**else**{  
 *// Add node to line3* line3.Enqueue(nextUp.entryTime, nextUp.name, nextUp.transactionType, nextUp.transactions);  
  
 }  
}

*/\*  
  
 Interface for Queue  
 @author: Garrett Wells  
 @date: 3-1-19  
  
\*/*#ifndef QUEUE\_H  
#define QUEUE\_H  
  
#include **"linkedlist.cpp"  
  
class** Queue{  
**private**:  
 LinkedList list;  
  
**public**:  
 Queue(**void**); *// Create new queue*

*// Add customer to queue end of queue* **void** Enqueue(**int** entryTime, string name, **char** transactionType, **int** numTransactions);

Node Remove(**void**); *// Remove the customer at the end of the queue* Node Dequeue(**void**); *// Remove the customer at the front of the queue* Node Peek(**void**); *// Return a pointer to the Node at the front of the queue* **int** getSize(**void**); *// Return the number of Nodes in the list* **void** Print(**void**); *// Print details of all nodes in the list*};  
  
#endif

*/\*  
  
 Implementation file for queue  
 @author: Garrett Wells  
 @date: 3-2-19  
  
\*/*#include **"queue.h"***// Create new queue*Queue::Queue(**void**){  
}  
  
*// Add customer to queue***void** Queue::Enqueue(**int** entryTime, string name, **char** transactionType,

**int** numTransactions){

list.AddToEnd(entryTime, name, transactionType, numTransactions);

}  
  
*// Remove the customer at the end of the queue*Node Queue::Remove(**void**){  
 **return** list.RemoveFromBack();  
}  
  
 *// Remove the customer at the front of the queue*Node Queue::Dequeue(**void**){  
 **return** list.RemoveFromFront();  
}  
  
*// Return the number of Nodes in the list***int** Queue::getSize(**void**) {  
 **return** list.getSize();  
}  
  
*// Call LinkedList.Print()***void** Queue::Print(**void**){  
 **if**(list.getSize() != 0){  
 list.Print();  
 }  
}  
  
*// Return the first node in the line*Node Queue::Peek(**void**) {  
 **return** list.Peek();  
}

/\*

Linked List Interface file

@author: Garrett Wells

@date: 3-4-19

\*/

#ifndef LINKEDLIST\_H  
#define LINKEDLIST\_H  
  
#include **<string>**#include **<iostream>  
using namespace** std;  
  
*//----------***struct** Node{  
 string name; *// Name of the customer* **int** transactions; *// Number of transactions the customer is conducting* **char** transactionType; *// Checking, Deposit, New Account, Reconcile Account* **int** entryTime; *// Entry time of customer* **struct** Node\* next; *// Pointer to the next node in the list*};  
  
**typedef struct** Node Node;  
**typedef** Node\* Nodeptr;  
  
*//----------***class** LinkedList{  
**private**:  
 Nodeptr head; *// First Node in the list* **int** size; *// Number of elements in the list***public**:  
 LinkedList(**void**); *// Creates new empty list* **void** AddToEnd(**int** entryTime, string name, **char** transactionType, **int** numTransactions); *// Adds node to end of list* **void** AddToFront(**int** entryTime, string name, **char** transactionType, **int** numTransactions); *// Adds node to front of list* Node RemoveFromFront(**void**); *// Remove the node from the front, return name* Node RemoveFromBack(**void**); *// Remove the node from the back and return name* **void** Print(**void**); *// Print contents of list* Node Peek(**void**); *// Return a pointer to the first Node in the list* **int** getSize(**void**); *// Return the number of Nodes in the list*};  
  
#endif

*/\*  
  
 Implementation file for LinkedList  
 @author: Garrett Wells  
 @date: 2-24-19  
  
\*/*#include **"linkedlist.h"** *// Creates new empty list*LinkedList::LinkedList(**void**){  
 head = NULL;  
 size = 0;  
}  
  
*// Adds node to end of list***void** LinkedList::AddToEnd(**int** entryTime, string name, **char** transactionType,

**int** numTransactions)

{  
  
 Nodeptr ptr = **new** Node();  
 ptr->entryTime = entryTime;  
 ptr->name = name;  
 ptr->transactions = numTransactions;  
 ptr->transactionType = transactionType;  
 ptr->next = NULL;  
  
 **if**(head == NULL){  
 head = ptr;  
 }**else**{  
 Nodeptr itr = head;  
  
 **while**(itr->next != NULL){  
 itr = itr->next;  
 }  
  
 itr->next = ptr;  
 }  
  
 size++;  
}  
  
*// Adds node to front of list***void** LinkedList::AddToFront(**int** entryTime, string name, **char** transactionType,

**int** numTransactions)

{  
 Nodeptr ptr = **new** Node();  
 ptr->entryTime = entryTime;  
 ptr->name = name;  
 ptr->transactions = numTransactions;  
 ptr->transactionType = transactionType;  
 ptr->next = NULL;  
  
 **if**(head == NULL){  
 head = ptr;  
 }**else**{  
 ptr->next = head;  
 head = ptr;  
 }  
  
 size++;  
}  
  
*// Remove the node from the front, return name*Node LinkedList::RemoveFromFront(**void**){  
 **if**(head != NULL){  
 Nodeptr ptr = head;  
 head = head->next;  
  
 Node temp = \*ptr; *// Save value in the Node* **delete** ptr; *// Delete the Node pointer* size--;  
  
 **return** temp;  
  
 }**else**{  
 size = 0;  
 cout << **"[ERROR]: empty list"** << endl;  
 **return** \*head;  
 }  
}  
  
*// Remove the node from the back and return name*Node LinkedList::RemoveFromBack(**void**){  
 **if**(head != NULL){  
 *// Traverse list and return last Node* Nodeptr ptr = head;  
 Nodeptr itr = head;  
  
 **while**(ptr->next != NULL){  
 itr = ptr;  
 ptr = ptr->next;  
 }  
  
 Node temp = \*ptr; *// Save values stored by last Node* **delete** ptr; *// Delete the Node* itr->next = NULL; *// Break link* size--;  
  
 **return** temp;  
  
 }**else**{  
 size = 0;  
 cout << **"[ERROR]: empty list"** << endl;  
 **return** \*head;  
 }  
}  
  
*// Returns head*Node LinkedList::Peek(**void**){  
 **return** \*head;  
}  
  
*// Returns the number of Nodes in the list***int** LinkedList::getSize(**void**){  
 **return** size;  
}  
  
**void** LinkedList::Print(**void**){  
 **if**(head != NULL){  
 Nodeptr itr = head;  
  
 **while**(itr->next != NULL){  
 cout << itr->name << endl  
 << **"Entry Time: "** << itr->entryTime << endl  
 << **"Transactions: "** << itr->transactions << endl  
 << **"Transaction Type: "** << itr->transactionType << endl;  
  
 itr = itr->next;  
 }  
  
 cout << itr->name << endl  
 << **"Entry Time: "** << itr->entryTime << endl  
 << **"Transactions: "** << itr->transactions << endl  
 << **"Transaction Type: "** << itr->transactionType << endl;  
  
 }**else**{  
 cout << **"\n"**;  
 }  
}

/\*

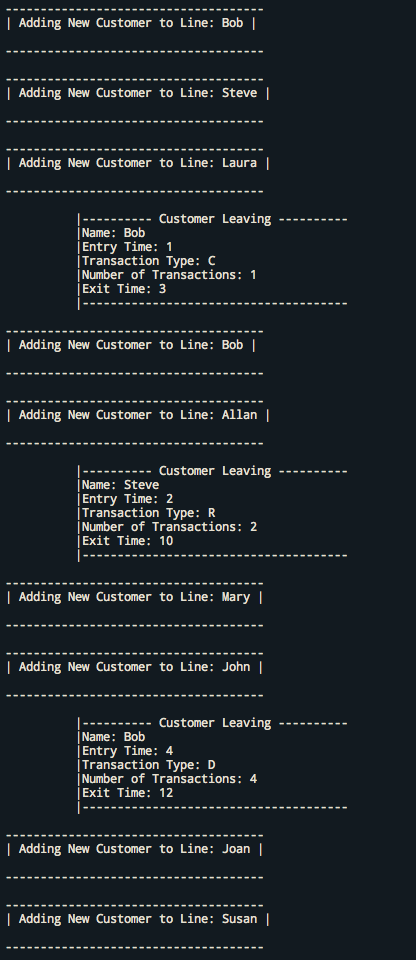
Tester Interface and Implementation file

@author: Garrett Wells

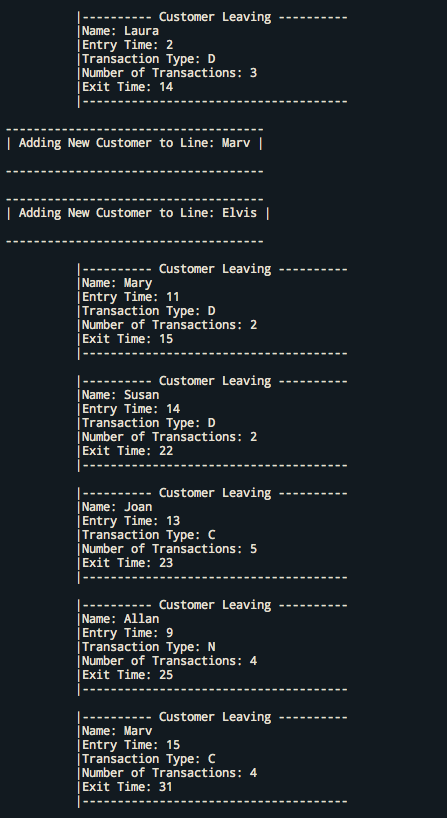
@date: 3-4-19

\*/

#ifndef TESTER\_H  
#define TESTER\_H  
  
**class** Tester{  
**private**:  
 **int** numTestsPassed;  
 **int** numTests;  
  
**public**:  
 Tester(**void**);  
  
 **int** testQueue(**void**);  
 **int** testLinkedList(**void**);  
};  
  
Tester::Tester(**void**){  
 numTests = 3;  
 numTestsPassed = 0;  
  
 numTestsPassed += testLinkedList();  
 numTestsPassed += testQueue();  
 numTestsPassed += testBankSim();  
  
 std:cout << **"----- Test Results -----"** << endl  
 << **"Tests Passsed: "** << numTestsPassed << **"/"** << numTests << endl;  
}  
  
**int** Tester::testLinkedList(**void**){  
 LinkedList list;  
  
 list.AddToEnd(1, **"adam"**, **'C'**, 2);  
 list.AddToEnd(2, **"barry"**, **'D'**, 1);  
 list.AddToEnd(3, **"calum"**, **'N'**, 3);  
  
 list.Print();  
  
 Node temp;  
  
 temp = list.RemoveFromFront();  
 **if**(temp.name != **"adam"**)  
 {  
 **return** 0;  
 }  
  
 temp = list.RemoveFromBack();  
 **if**(temp.name != **"calum"**)  
 {  
 **return** 0;  
 }  
  
 temp = list.RemoveFromFront();  
 **if**(temp.name != **"barry"**)  
 {  
 **return** 0;  
 }  
  
 **return** 1;  
}  
  
**int** Tester::testQueue(**void**){  
 *// Create new Queue of customers and remove them and print their data* Node ptr;  
 Queue line;  
  
 line.Enqueue(1, **"adam"**, **'C'**, 2);  
 line.Enqueue(2, **"barry"**, **'D'**, 1);  
 line.Enqueue(3, **"calum"**, **'N'**, 3);  
  
 cout << **"---------- Printing Queue ----------"** << endl;  
 *// Print Queue contents* line.Print();  
  
 cout << **"---------- Dequeueing Contents ----------"** << endl;  
 *// Dequeue contents and print in order* ptr = line.Dequeue();  
 **if**(ptr.name != **"adam"**){**return** 0;}  
  
 ptr = line.Dequeue();  
 **if**(ptr.name != **"barry"**){**return** 0;}  
  
 ptr = line.Dequeue();  
 **if**(ptr.name != **"calum"**){**return** 0;}  
  
 cout << **"---------- Dequeueing Successful ----------"** << endl;  
  
 **return** 1;  
}  
  
#endif

****

**Program Output**

****