**CS311 For your HW5 – Required Functions**

**Handle the error case, special case and then the regular case in this order.**

**You must comment each if/else with what case it is.**

**You must make sure any new Rear node has ->Next = NULL.**

**Constructor** - initialize **Front** and **Rear** to be NULL and **Count** = 0.

This does not create any node. The list is empty.

Please place a cout as the first line of this function. “calling the llist constructor”

We will need it in preparation for HW6.

**Destructor** - while the list is not empty, call **deleteFront** repeatedly to delete all nodes.

Please place a cout as the first line of this function. “calling the llist destructor”

We will need it in preparation for HW6.

**boolean isEmpty()**

return true if Front **and** Rear are both pointing to NULL **and** Count is 0.

**void displayAll()**

display each element of the list starting from Front, enclosed in [ ] followed **by Count** and endl.

E.g. [ 1 2 3 ] 3 elements

(**special case:** if the list is empty, display [ empty ] ).

**void addRear(el\_t NewNum) – 2 cases**

**special case:** if this is going to be the very first node, you must

create a new node and make Front and Rear point to it. Put NewNum in the Elem field

of this new node. Count is updated.

**regular case:** add a new node at the rear and puts NewNum in the Elem field

of this new node. Count is updated.

Regular:

**Rear->Next = new Node;**

**Rear = Rear->Next;**

**Rear->Elem = NewNum;**

**Rear->Next = NULL;**

**void deleteFront(el\_t& OldNum) – 3 cases**

**error case:** if the List is empty, throw an exception.

**special case:** if this is going to make the list empty, give back the front node element through OldNum

(pass by reference) and also remove the front node. Count is updated.

Make sure both Front and Rear both become NULL.

**regular case:** give back the front node element through OldNum (pass by reference)

and also remove the front node. Count is updated.

Regular:

**OldNum = Front->Elem;**

**Node \*Second;**

**Second = Front->Next;**

**delete Front;**

**Front = Second;**

**void addFront(el\_t NewNum) – 2 cases**

**special case:** if this is going to be the very first node, you must

create a new node and make Front and Rear point to it. Put NewNum in the Elem field

of this new node. Count is updated.

**regular case:** add a new node at the front and put NewNum in the Elem field

of this new node. Count is updated.

**void deleteRear(el\_t& OldNum) – 3 cases**

**error case:** if empty, throw an exception.

**special case:** if this is going to make the list empty,

give back the rear node element through OldNum (pass by reference)

and also remove the rear node. Count is updated.

Make sure both Front and Rear become NULL).

**regular case:** give back the rear node element through OldNum (pass by reference)

and also remove the rear node. Count is updated.

**void deleteIth(int I, el\_t& OldNum) – 4 cases**

**Error cases:** If I is an illegal value (i.e. > Count or < 1) throw an exception.

**Special cases:** this should simply call deleteFront (I = 1) or deleteRear (I == Count) for some cases.

**Regular case:** will delete the Ith node (**I starts out at 1**). Count is updated.

**Note that if you go to the I-1th node,**

**and also place another pointer on the I+1th node,**

**you can bypass the Ith node. Draw pictures first.**

**void insertIth(int I, el\_t newNum) – 4 cases**

**Error cases:** If I is an illegal value (i.e. > **Count+1** or < 1) throw an exception.

**Special cases**: this should simply call addFront (I == 1) or addRear (I == Count+1) for some cases.

**Regular case:** will add right before the Ith node. Count is updated.

**Note that if you go to the I-1th node, and also place another pointer on the Ith node,**

**you can place a new node between them.**

**Draw pictures first.**

**copy constructor for pass by value and return by value (see below)**

**operator= for allowing the client to use = for linked list objects (see below)**

**Allow L1 = L2 in the client**

**llist& llist::operator=(const llist& OtherOne)**

**{**

**// First make sure this-> and OtherOne are not the same object.**

**// To do this, compare the pointers to the objects .**

**if (&OtherOne != this) // if not the same**

**{**

**// this-> object has to be emptied first.**

**while (! this->isEmpty() )**

**this->deleteRear();**

**// this-> object has to be built up by allocating new cells with OtherOne elements (\*\*)**

**Node\* P; // local pointer for OtherOne**

**P = OtherOne.Front;**

**while (P != NULL)** **// a loop which repeats until you reach the end of OtherOne.**

**{**

**this->addRear(P->elem); //P’s element is added to this->**

**P = P->Next; // Go to the next node in OtherOne**

**}**

**}// end of if**

**return \*this; // return the result unconditionally. Note that the result is returned by reference.**

**}**

**Allow passing of a list by value and returning of a list by value**

**// Note that the Original is being passed to the constructor by reference**

**// to create a new object this-> as a copy of the Original**

**llist::llist(const llist& Original)**

**{**

**// this->'s data members need to be initialized here first**

**Front = NULL; Rear = NULL; Count = 0;**

**// this-> object has to be built up by allocating new cells**

**// and copying the values from Original into each cell as we did with**

**// operator= above. To do this,**

**copy here the (\*\*) lines in Operator Overloading of = but note that it is Original and not OtherOne.**

**// Nothing to return because this is a constructor.**

**}**

**Make sure you use the Loop Templates from the notes.**