```
// Stefan Retief
// CS211 DLL, HW7
// DLL Class - Class File
// ==================================================

#include <iostream>
#include "DLL.hpp"

//PURPOSE:      Constructor which initilizes front, rear, and count
DLL::DLL() {
    front = NULL;
    rear = NULL;
    count = 0;
}

//PURPOSE:      When the program hits and endbrace '}', delete the DDLL
//ALGORITHM:    While the LQueue isn't empty, delete front
DLL::~DLL() {
    while (!isEmpty()) {
        deleteFront();
    }
}

//PURPOSE:      To add en element to the rear of the lqueue
//PARAMS:       the element to add the the lqueue
//ALGORITHM:    if empty, make a new node and set the element. front->next = NULL
//              else, make a new node on rear->next, set the element and set rear
//              ->next to NULL
void DLL::addRear(el_t value) {
    if (isEmpty()) {
        front = new node;
        rear = front;
        front->elem = value;
        front->next = NULL;
        front->prev = NULL;
        count++;
    }
    else {
        rear->next = new node;
        rear->next->prev = rear;
        rear = rear->next;
        rear->elem = value;
        rear->next = NULL;

        count++;
    }
}

//PURPOSE:      To delete the front pointer and display the element
//PARAMS:       None
//ALGORITHM:    while the lqueue isn't empty, set a temp element to front->next
//              and deletethe front. Set the temp to the front and return the
    element
el_t DLL::deleteFront() {
    if (isEmpty()) {
        queueError("UNABLE TO REMOVE FRONT. QUEUE UNDERFLOW.");
```

```
        return 0;
    }

    else {
        if (count == 1) {
            el_t second = front->elem;
            delete front;
            front = NULL;
            rear = NULL;
            count--;
            return second;
        }
        else {
            if (isEmpty()) {
                queueError("UNABLE TO REMOVE REAR. QUEUE UNDERFLOW");
                return 0;
            }
            else {
                node* second = front->next;
                el_t sec = front->elem;
                delete front;
                front = second;
                front->prev = NULL;
                count--;
                return sec;
            }
        }
    }
}

//PURPOSE:      To return if a queue is empty or not (returns true or false)
//PARAMS:       None
//ALGORITHM:    if count is at the initilized state of 0, return true, else
//              return false
bool DLL::isEmpty() {
    if (front == NULL && rear == NULL)
        return true;
    else
        return false;
}

//PURPOSE:      To display all the elements in the lqueue
//PARAMS:       None
//ALGORITHM:    While the lqueue isn't empty, put a temp pointer to the front
//              and while the temp->next = NULL, keep displaying and moving
//              the element
void DLL::displayAll() {
    if (isEmpty()) {
        cout << "[EMPTY]" << endl;
    }
    else {
        node* p = front;
        while (p != NULL) {
            cout << "[";
            cout << p->elem;
            p = p->next;
```

```
            cout << "]" << " ";
        }
        cout << endl;
    }
}

//PURPOSE:      To display all the elements in the DLL in reverse
//PARAMS:       None
//ALGORITHM:    While the lqueue isn't empty, put a temp pointer to the rear
//              and while the temp->next = NULL, keep displaying and moving
//              the element
void DLL::printAllReverseDLL() {
    if (isEmpty()) {
        cout << "[EMPTY]" << endl;
    }
    else {
        node* p = rear;
        while (p != NULL) {
            cout << "[";
            cout << p->elem;
            p = p->prev;
            cout << "]" << " ";
        }
        cout << endl;
    }
}

//PURPOSE:      To add an node to the front before aDLL other pointers
//PARAMS:       The element you would like to add
//ALGORITHM:    stores the front node in a temp pointer, makes a new pointer
//              in front and sets front->next to the temp pointer
void DLL::addFront(el_t elem) {
    if (isEmpty()) {
        addRear(elem);
    }
    else {
        front->prev = new node;
        front->prev->next = front;
        front->prev->elem = elem;
        front->prev->prev = NULL;
        front = front->prev;
        count++;
    }
}

//PURPOSE:      To delete the rear pointer from the DLL
//PARAMS:       None
//ALGORITHM:    determines if the DLL has one, none or many elements, then
//              stores the element in a temp variable, sets the rear pointer to
//              NULL and the previous become rear, returns the element
el_t DLL::deleteRear() {

    if (isEmpty()) {
        queueError("UNABLE TO REMOVE REAR. QUEUE UNDERFLOW");
        return 0;
    }
```

```
    else {
        if (count == 1) {
            el_t elem = rear->elem;
            delete rear;
            front = NULL;
            rear = NULL;
            count--;
            return elem;
        }
        else {
            if (isEmpty()) {
                queueError("UNABLE TO REMOVE REAR. QUEUE UNDERFLOW");
                return 0;
            }
            else {
                rear = rear->prev;
                el_t elem = rear->next->elem;
                delete rear->next;
                rear->next = NULL;
                count--;
                return elem;
            }
        }
    }
}

//PURPOSE:      To delete the first node containing the element
//PARAMS:       The element to search for and delete
//ALGORITHM:    makes two pointers, and starts from the front and moves back
//              if the elem is found, delete the node, set pre->next to the
//              node foDLLowing delete and display message
void DLL::deleteNode(el_t e) {

    if (!isEmpty()) {
        if (front->elem == e)
            deleteFront();
        else {
            node* del;
            for(del = front->next; del!= NULL && del->elem != e; del = del->next)
                ;

            if(del != NULL) {
                if (del == rear)
                    deleteRear();
                else {
                    del->prev->next = del->next;
                    del->next->prev = del->prev;
                    delete del;
                    count--;
                }
            }
        }
    }
}

//PURPOSE:      To delete aDLL nodes containing the element
```

```
//PARAMS:       The element to search for and delete
//ALGORITHM:    Makes two pointers, and searches until del == NULL. If
//              Elem is found, delete and makes pre->next the node following del
//              displays message if found or not.
void DLL::deleteNodes(el_t e) {

    if (!isEmpty()) {

        node* del = front->next;
        while (del != NULL) {
            if (del->elem == e) {
                if (del == rear)
                    deleteRear();
                else {
                    del->prev->next = del->next;
                    del->next->prev = del->prev;
                    delete del;
                    count--;
                }
            }
            del = del->next;
        }
        if (front->elem == e)
            deleteFront();
    }
}

//PURPOSE:      (Private) To handle unexpected errors encountered by other
//              methods
//PARAMS:       String message to be displayed
//ALGORITHM:    'cout' the message and exit program with error code 1
void DLL::queueError(string msg) {
    cout << msg << endl;
    exit(1);
}

//PURPOSE:      To add the elements in order from Low to High
//PARAMS:       the element to add
//ALGORITHM:    checks the value of other elements before adding the element in
    the
//              correct order
void DLL::addInOrderAscend(el_t e) {

    node* p = front;

    if (isEmpty() || front->elem > e)
        addFront(e);
    else {
        while (p->next != NULL && p->next->elem < e)
            p = p->next;

        if (p == rear && p->elem < e)
            addRear(e);
        else {
            node* savedNext = p->next;
            p->next = new node;
```

```
            p->next->elem = e;
            p->next->prev = p;
            p->next->next = savedNext;
                savedNext->prev = p->next;
            count++;
        }
    }
}

//PURPOSE:      To add the elements in order from high to low
//PARAMS:       the element to add
//ALGORITHM:    checks the value of other elements before adding the element in
    the
//              correct order
void DLL::addInOrderDescend(el_t e) {

    node* p = front;

    if (isEmpty() || front->elem < e)
        addFront(e);
    else {
        while (p->next != NULL && p->next->elem > e)
            p = p->next;

        if (p == rear && p->elem > e)
            addRear(e);
        else {
            node* savedNext = p->next;
            p->next = new node;
            p->next->elem = e;
            p->next->prev = p;
            p->next->next = savedNext;
            savedNext->prev = p->next;
            count++;
        }
    }
}

//PURPOSE:      To search for the element
//PARAMS:       The element to find
//ALGORITHM:    Check the element of each list until it is found or not
bool DLL::search(el_t e) {
    if (isEmpty()) {        //is empty, it's false
        return false;
    }
    else {
        node* scan = front; //start from the front
        while(scan != NULL) {       //while we don't get to the end
            if (scan->elem == e) {  //if elem == e, return true
                return true;
            }
            scan = scan->next;
        }
    }
    return false;
}
```