Listing 1: DLL.hpp

```
#pragma once

#include <doctest.h>
#include <iostream>
#include <stdexcept>
#include <sstream>

/*----------------------------------------------------------------------------
 * class definition
 *----------------------------------------------------------------------------*/

/**
 * @brief CMP 246 Module 5 generic doubly-linked list, supporting iterators.
 *
 * DLL is a generic doubly-linked list data structure. It allows inserting at
 * the front or back of the list, and supports index-based get, set, and remove
 * operations. The list also provides a contains method, and the administrative
 * methods clear, isEmpty, and size. DLL also has a copy constructor, and
 * overrides the assignment and stream insertion operators. DLL provides front,
 * back, and end methods to access iterators that move through the list from
 * front to back or back to front.
 */
template <class T> class DLL {

private:
    /**
     * @brief Node in the doubly-linked list.
     *
     * Node is a private inner class of DLL. The class represents a
     * single node in the list. Each node has a payload of type T, a
     * pointer to the next node in the list, and a pointer to the previous node
     * in the list.
     */
    class Node {
    public:
        /**
         * @brief Default constructor.
         *
         * Make a new Node with default data and next and previous pointers set
         * to zero.
         */
        Node() : data(), pPrev(0), pNext(0) { }

        /**
         * @brief Initializing constructor.
         *
         * Make a new node with the specified data and previous and next
         * pointer values.
         *
         * @param d Data value for the node.
         * @param pP Pointer to the previous node in the list, or 0 if this is
         * the first node in the list.
         * @param pN Pointer to the next node in the list, or 0 if this is the
         * last Node in the list.
         */
        Node(const T &d, Node *pP, Node *pN) : data(d), pPrev(pP), pNext(pN) { }

        /**
         * @brief Node payload.
         *
         * Type T payload of the node. Assumed to support assignment, equality
         * testing, copy constructor, and stream insertion.
         */
        T data;

        /**
         * @brief Previous node pointer.
         *
         * Pointer to the previous node in the list, or 0 if this is the first
         * node.
         */
        Node *pPrev;

        /**
```

```
         * @brief Next node pointer.
         *
         * Pointer to the next node in the list, or 0 if this is the last node.
         */
        Node *pNext;
    };

public:
    /**
     * @brief DLL iterator.
     *
     * This class allows DLL users to iterate through the list, from front to
     * back or back to front, without exposing the pointer structure of the
     * list, and without incurring the time complexity of successive calls to
     * the index-based get() and set() methods.
     */
    class Iterator {
    public:
        /**
         * @brief Iterator dereferencing operator.
         *
         * This override of the dereferencing operator allows DLL
         * users to access and / or change the payload of a node in the list.
         *
         * @throws std::out_of_range if the iterator is past either end of
         * the list.
         */
        T &operator*();

        /**
         * @brief Iterator equality operator.
         *
         * This override of the equality operator allows DLL users to
         * compare two iterators, to determine if they refer to the same node
         * in the list.
         */
        bool operator==(const Iterator &other) const {
            return pCurr == other.pCurr;
        }

        /**
         * @brief Interator inequality operator.
         *
         * This override of the inequality operator allows DLL users to
         * compare two iterators, to determine if they refer to different nodes
         * in the list.
         */
        bool operator!=(const Iterator &other) const {
            return pCurr != other.pCurr;
        }

        /**
         * @brief Iterator decrement operator.
         *
         * This override of the postfix decrement operator allows DLL
         * users to move an iterator from one node to the previous node in the
         * list.
         *
         * @param dummy Dummy parameter that indicates we are overriding
         * the postfix decrement operator.
         *
         * @throws std::out_of_range if the iterator is past either end of
         * the list.
         */
        Iterator &operator--(int dummy);

        /**
         * @brief Iterator increment operator.
         *
         * This override of the postfix increment operator allows DLL
         * users to move an iterator from one node to the next node in the
         * list.
         *
         * @param dummy Dummy parameter that indicates we are overriding
         * the postfix increment operator.
         *
         * @throws std::out_of_range if the iterator is past either end of
         * the list.
         */
        Iterator &operator++(int dummy);

        // Make DLL a friend class, so it can access the private constructor
        friend class DLL;

    private:
        /**
         * @brief Initializing constructor.
         *
         * This constructor makes an iterator that refers to the Node at the
         * end of the specified pointer. The constructor is private, so that
         * the only way to get an iterator is via the DLL front(), back(),
         * and end() methods.
         *
         * @param pC Node this iterator should refer to.
         */
        Iterator(Node *pC) : pCurr(pC) { }

        /**
         * Pointer to the Node this iterator refers to.
         */
        Node *pCurr;
    };

    /**
     * @brief Default list constructor.
     *
     * Made an initially empty list.
     */
    DLL() : pHead(0), pTail(0), n(0u) { }

    /**
     * @brief Copy construstor.
     *
     * Make a new, deep-copy list, just like the parameter list.
     *
     * @param otherList Reference to the DLL to copy.
     */
    DLL(const DLL<T> &otherList) : pHead(0), pTail(0), n(0u) { copy(otherList); }

    /**
     * @brief Destructor.
     *
     * Free the memory used by this list.
     */
    ~DLL() { clear(); }

    /**
     * @brief Add a value to the front of the list.
     *
     * @param d Value to add to the list.
     */
    void addFirst(const T &d);

    /**
     * @brief Add a value to the back of the list.
     *
     * @param d Value to add to the list.
     */
    void addLast(const T &d);

    /**
     * @brief Get an Iterator on the last node of the list.
     *
     * @return An Iterator positioned on the last node of the list.
     */
    Iterator back() const { return Iterator(pTail); }

    /**
     * @brief Clear the list.
     *
     * Remove all the elements from the list.
     */
    void clear();

    /**
     * @brief Search the list for a specified value.
     *
     * Searches for a value and returns the index of the first occurrence
     * of the value in the list, or -1 if the value is not in the list.
     *
     * @param d Value to search for.
     *
     * @return Index of the first occurrence of d in the list, or -1 if it is
     * not in the list.
     */
    int contains(const T &d) const;

    /**
     * @brief Get an Iterator representing the end of the list.
     *
     * @return An Iterator representing one past the back, or one before the
     * front, of the list.
     */
    Iterator end() const { return Iterator(0); }

    /**
     * @brief Get an Iterator on the first node of the list.
     *
     * @return An Iterator positioned on the first node of the list.
     */
    Iterator front() const { return Iterator(pHead); }

    /**
     * @brief Get a value.
     *
     * Get the value at a specified index in the list.
     *
     * @param idx Index of the value to get.
     *
     * @throws std::out_of_range if the index is past either end of the list.
     *
     * @return Value at location idx in the list.
     */
    T get (unsigned idx) const;

    /**
     * @brief Get first value.
     *
     * Get the value at the front of the list.
     *
     * @throws std::out_of_range if the list is empty.
     *
     * @return Value at the front of the list.
     */
    T getFirst() const;

    /**
     * @brief Get last value.
     *
     * Get the value at the back of the list.
     *
     * @throws std::out_of_range if the list is empty.
     *
     * @return Value at the back of the list.
     */
    T getLast() const;

    /**
     * @brief Determine if the list is empty.
     *
     * Convenience method to test if the list contains no elements.
     *
     * @return true if the list is empty, false otherwise.
     */
    bool isEmpty() const { return size() == 0u; }

    /**
     * @brief Remove an element.
     *
     * Remove the value at a specified index in the list.
     *
     * @param idx Index of the element to remove.
     *
     * @throws std::out_of_range if the index is past either end of the list.
     *
     * @return Value that was at location idx.
     */
    T remove(unsigned idx);

    /**
     * @brief Remove first element.
     *
     * Remove the element from the front of the list.
     *
     * @throws std::out_of_range if the list is empty.
     *
     * @return Value that was at the front of the list.
     */
    T removeFirst();

    /**
     * @brief Remove last element.
     *
     * Remove the element from the back of the list.
     *
     * @throws std::out_of_range if the list is empty.
     *
     * @return Value that was at the back of the list.
     */
    T removeLast();

    /**
     * @brief Change a list element.
     *
     * Change the value at a specified index to another value.
     *
     * @param idx Index of the value to change.
     *
     * @throws std::out_of_range if the index is past either end of the list.
     *
     * @param d New value to place in position idx.
     */
    void set(unsigned idx, const T &d);

    /**
     * @brief Change the first list element.
     *
     * @param d New value to replace front value in the list.
     *
     * @throws std::out_of_range if the list is empty.
     */
    void setFirst(const T &d);

    /**
     * @brief Change the last list element.
     *
     * @param d New value to replace back value in the list.
     *
     * @throws std::out_of_range if the list is empty.
     */
    void setLast(const T &d);

    /**
     * @brief Get list size.
     *
     * Get the number of integers in the list.
     *
     * @return The number of integers in the list.
     */
    unsigned size() const { return n; }

    /**
     * @brief Assignment operator.
     *
     * Override of the assignment operator to work with DLL objects.
     * Makes this list a deep-copy, identical structure as the parameter
     * DLL.
     *
     * @param list DLL to copy from
     *
     * @return Reference to this object.
     */
    DLL<T> &operator=(const DLL<T> &otherList);

private:
    /**
     * @brief Copy helper method.
     *
     * This private helper method is used to deep-copy all of the elements from
     * the parameter list to this list. Any existing elements in this list are
     * safely removed before the copy.
     *
     * @param otherList Reference to the DLL object to copy from.
     */
    void copy(const DLL<T> &otherList);

    /**
     * Pointer to the first Node in the list, or 0 if the list is empty.
     */
    Node *pHead;

    /**
     * Pointer to the last Node in the list, or 0 if the list is empty.
     */
    Node *pTail;

    /**
     * Number of integers in the list.
     */
    size_t n;
};

//-----------------------------------------------------------------------------
// function implementations
//-----------------------------------------------------------------------------

/*
 * Iterator dereferencing operator override.
 */
template <class T>
T &DLL<T>::Iterator::operator*() {
    // if the iterator is past either end of the list, throw an exception
    if(pCurr == 0) {
        throw std::out_of_range("Dereferencing␣beyond␣list␣end␣in␣"
                                "Iterator::*()");
    }

    return pCurr->data;
}

// doctest unit test for iterator dereferencing
TEST_CASE("testing␣DLL<T>::Iterator␣dereferencing") {
    DLL<int> list;

    list.addFirst(1);
    list.addFirst(0);
    DLL<int>::Iterator it = list.front();

    // first element should be 0
    CHECK(*it == 0);

    // last element should be 1
    it++;
    CHECK(*it == 1);

    // check exception handling when dereferencing past end of list
    it++;
    bool flag = true;
    try {
        *it;            // should throw an exception
        flag = false;   // this should never happen
    } catch (std::out_of_range oor) {
        CHECK(flag);
    }
}

/*
 * Iterator decrement operator overload.
 */
template <class T>
typename DLL<T>::Iterator &DLL<T>::Iterator::operator--(int dummy) {
    // if the iterator is past the end of the list, throw an exception
    if(pCurr == 0) {
        throw std::out_of_range("Increment␣beyond␣list␣end␣in␣"
                                "Iterator::--()");
    }

    pCurr = pCurr->pPrev;
    return *this;
}

// doctest unit test for iterator decrement overload
TEST_CASE("testing␣DLL<T>::Iterator␣postfix␣decrement") {
    DLL<char> list;

    // populate with a - z
    for(char c = ’a’; c <= ’z’; c++) {
        list.addFirst(c);
    }

    // verify that iterating moves through the list backwards
    DLL<char>::Iterator it = list.back();
    for(char c = ’a’; c <= ’z’; c++) {
        CHECK(*it == c);
        it--;
    }

    // check exception handling when incrementing beyond list end
    bool flag = true;
    try {
        it--;           // this should throw an exception
        flag = false;   // this should never happen
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

/*
 * Iterator increment operator overload.
 */
template <class T>
typename DLL<T>::Iterator &DLL<T>::Iterator::operator++(int dummy) {
    // if the iterator is past the end of the list, throw an exception
    if(pCurr == 0) {
        throw std::out_of_range("Increment␣beyond␣list␣end␣in␣"
                                "Iterator::++()");
    }

    pCurr = pCurr->pNext;
    return *this;
}

// doctest unit test for iterator increment overload
TEST_CASE("testing␣DLL<T>::Iterator␣postfix␣increment") {
    DLL<char> list;

    // populate with a - z
    for(char c = ’a’; c <= ’z’; c++) {
        list.addFirst(c);
    }

    // verify that iterating moves through the list
    DLL<char>::Iterator it = list.front();
    for(char c = ’z’; c >= ’a’; c--) {
        CHECK(*it == c);
        it++;
    }

    // check exception handling when incrementing beyond list end
    bool flag = true;
    try {
        it++;           // this should throw an exception
        flag = false;   // this should never happen
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

// doctest unit test for the copy constructor
TEST_CASE("testing␣DLL<T>␣copy␣constructor") {
    DLL<int> list1;

    // populate the original list
    for(int i = 0; i < 5; i++) {
        list1.addFirst(i);
    }

    // make a new list like original
    DLL<int> list2(list1);

    // does it have the right size?
    CHECK(list2.size() == list1.size());

    // does it have the right elements?
    for(int i = 0; i < 5; i++) {
        CHECK(list2.get(i) == (4 - i));
    }

    // try it again with dynamic allocation
    DLL<int> *pList = new DLL<int>(list1);

    // does it have the right size?
    CHECK(pList->size() == list1.size());

    // does it have the right elements?
    for(int i = 0; i < 5; i++) {
        CHECK(pList->get(i) == (4 - i));
    }

    delete pList;
}

/*
 * Add d to the front of the list.
 */
template <class T>
void DLL<T>::addFirst(const T &d) {
    Node *pN = new Node(d, 0, pHead);

    if(pHead == 0) {
        // empty list case
        pHead = pTail = pN;
    } else {
        // non-empty list case
        pHead = pN;
        pHead->pNext->pPrev = pHead;
    }

    n++;
}

// doctest unit test for addFirst
TEST_CASE("testing␣DLL<T>::addFirst") {
    DLL<int> list;

    list.addFirst(0);

    // is it there?
    DLL<int>::Iterator it = list.front();
    CHECK(*it == 0);

    // try with another element
    list.addFirst(1);
    it = list.front();
    CHECK(*it == 1);
    it = list.back();
    CHECK(*it == 0);
}

/*
 * Add do the back of the list.
 */
template <class T>
void DLL<T>::addLast(const T &d) {
    Node *pN = new Node(d, pTail, 0);

    if(pHead == 0) {
        // empty list case
        pHead = pTail = pN;
    } else {
        // non-empty list case
        pTail = pN;
        pTail->pPrev->pNext = pTail;
    }

    n++;
}

// doctest unit test for addLast
TEST_CASE("testing␣DLL<T>::addLast") {
    DLL<int> list;

    list.addLast(0);

    // is it there?
    DLL<int>::Iterator it = list.back();
    CHECK(*it == 0);

    // try with another element
    list.addLast(1);
    it = list.back();
    CHECK(*it == 1);
    it = list.front();
    CHECK(*it == 0);
}

// doctest unit test for the back method
TEST_CASE("testing␣DLL<T>::back") {
    DLL<int> list;
    list.addLast(0);

    // is the back iterator the first element?
    DLL<int>::Iterator it = list.back();
    CHECK(*it == list.get(0));

    // add another and repeat the check
    list.addLast(1);
    it = list.back();
    CHECK(*it == list.get(1));
}

/*
 * Delete all list nodes.
 */
template <class T>
void DLL<T>::clear() {
    // create cursors
    Node *pCurr = pHead, *pPrev = 0;

    // iterate thru list, deleting each node
    while(pCurr != 0) {
        // "inchworm" up to next node
        pPrev = pCurr;
        pCurr = pCurr->pNext;

        // delete previous node
        delete pPrev;
    }

    // reset head, tail pointer and size
    pHead = 0;
    pTail = 0;
    n = 0u;
}

// doctest unit test for the clear method
TEST_CASE("testing␣DLL<T>::clear") {
    DLL<int> list;

    // add some list elements
    for(int i = 0; i < 100; i++) {
        list.addFirst(i);
    }

    // clear should make size equal zero
    list.clear();
    CHECK(list.size() == 0u);
}

/*
 * Search the list for value d.
 */
template <class T>
int DLL<T>::contains(const T &d) const {
    // create cursors
    int idx = -1;
    Node *pCurr = pHead;

    // iterate until we find d or end of list
    while(pCurr != 0) {
        idx++;

        // found it? return its index
        if(pCurr->data == d) {
            return idx;
        }

        pCurr = pCurr->pNext;
    }

    // not found? return flag value
    return -1;
}

// doctest unit test for the contains method
TEST_CASE("testing␣DLL<T>::contains") {
    DLL<char> list;

    // populate the list
    for(char c = ’A’; c <= ’Z’; c++) {
        list.addFirst(c);
    }

    // search for 1st element in list
    CHECK(list.contains(’Z’) == 0);

    // search for last element in list
    CHECK(list.contains(’A’) == 25);

    // search for something in the middle
    CHECK(list.contains(’M’) == 13);

    // search for something not in list
    CHECK(list.contains(’a’) == -1);
}

/*
 * Make this list a deep copy of another list.
 */
template <class T>
void DLL<T>::copy(const DLL<T> &list) {
    // remove any existing data
    clear();

    // using iterator and addLast simplifies this method compared with the
    // equivalent in previous SLLs
    for(DLL<T>::Iterator i = list.front(); i != list.end(); i++) {
        addLast(*i);
    }
}

// since copy is private, it's tested indirectly in copy constructor and
// assignment operator tests

// doctest unit test for the end method
TEST_CASE("testing␣DLLT<T>::end") {
    DLL<double> list;

    // iterating through empty list should not happen
    DLL<double>::Iterator it = list.front();
    int count = 0;
    for(; it != list.end(); it++) {
        count++;
    }
    CHECK(count == 0);

    // iterating through a list w/ 5 elements
    for(int i = 0; i < 5; i++) {
        list.addFirst(i);
    }
    it = list.front();
    count = 0;
    for(; it != list.end(); it++) {
        count++;
    }
    CHECK(count == 5);
}

// doctest unit test for the front method
TEST_CASE("testing␣DLL<T>::front") {
    DLL<int> list;
    list.addFirst(0);

    // is the front iterator the first element?
    DLL<int>::Iterator it = list.front();
    CHECK(*it == list.get(0));

    // add another and repeat the check
    list.addFirst(1);
    it = list.front();
    CHECK(*it == list.get(0));
}

/*
 * Get the value at location idx.
 */
template <class T>
T DLL<T>::get(unsigned idx) const {
    // if the idx is past list end, throw an exception
    if(idx >= n) {
        throw std::out_of_range("Index␣out␣of␣range␣in␣DLL::get()");
    }

    // initialize cursor
    Node *pCurr = pHead;

    // iterate cursor to position
    for(unsigned i = 0u; i < idx; i++) {
        pCurr = pCurr->pNext;
    }

    // return requested value
    return pCurr->data;
}

// doctest unit test for the get method
TEST_CASE("testing␣DLL<T>::get") {
    DLL<char> list;

    // populate list
    for(char c = 'A'; c <= 'Z'; c++) {
        list.addFirst(c);
    }

    // get first element
    CHECK(list.get(0) == 'Z');

    // get last element
    CHECK(list.get(25) == 'A');

    // get something in the middle
    CHECK(list.get(13) == 'M');

    // check exception handling when access is beyond list
    bool flag = true;
    try {
        list.get(26); // list element 26 does not exist
        flag = false; // this line should not be reached, due to an exception
    } catch(std::out_of_range oor) {
        // verify flag wasn't modified
        CHECK(flag);
    }
}

/*
 * Get the front value.
 */
template <class T>
T DLL<T>::getFirst() const {
    // if list is empty, throw an exception
    if(pHead == 0) {
        throw std::out_of_range("Empty␣list␣in␣DLL::getFirst()");
    }

    return pHead->data;
}

// doctest unit test for the getFirst method
TEST_CASE("testing␣DLL<T>::getFirst") {
    DLL<int> list;
    for(int i = 0; i < 5; i++) {
        list.addFirst(i);
        CHECK(list.getFirst() == i);
    }

    // test exception generation
    list.clear();
    bool flag = true;
    try {
        list.getFirst();    // this should cause an exception
        flag = false;       // this should never happen
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

/*
 * Get the back value.
 */
template <class T>
T DLL<T>::getLast() const {
    // if list is empty, throw an exception
    if(pTail == 0) {
        throw std::out_of_range("Empty␣list␣in␣DLL::getLast()");
    }

    return pTail->data;
}

// doctest unit test for the getLast method
TEST_CASE("testing␣DLL<T>::getLast") {
    DLL<int> list;
    for(int i = 0; i < 5; i++) {
        list.addLast(i);
        CHECK(list.getLast() == i);
    }

    // test exception generation
    list.clear();
    bool flag = true;
    try {
        list.getLast();     // this should cause an exception
        flag = false;       // this should never happen
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

/*
 * Remove node at location idx.
 */
template <class T>
T DLL<T>::remove(unsigned idx) {
    // if the idx is past list end, throw an exception
    if(idx >= n) {
        throw std::out_of_range("Index␣out␣of␣range␣in␣DLL::remove()");
    }

    // handle special cases with other methods
    if(idx == 0u) {
        return removeFirst();
    } else if(idx == n - 1u) {
        return removeLast();
    }

    // handle the general case
    Node *pCurr = pHead;

    // iterate cursor to position
    for(unsigned i = 0u; i < idx; i++) {
        pCurr = pCurr->pNext;
    }

    // save value so we can return it
    T d = pCurr->data;

    // wire around the node to be removed
    pCurr->pPrev->pNext = pCurr->pNext;
    pCurr->pNext->pPrev = pCurr->pPrev;

    // remove node and decrement size
    delete pCurr;
    n--;

    // send back removed value
    return d;
}

// doctest unit test for the remove method
TEST_CASE("testing␣DLL<T>::remove") {
    DLL<char> list;

    // populate list
    for(char c = ’A’; c <= ’Z’; c++) {
        list.addFirst(c);
    }

    // remove first element
    CHECK(list.remove(0) == ’Z’);
    CHECK(list.size() == 25);
    CHECK(list.get(0) == ’Y’);

    // remove last element
    CHECK(list.remove(24) == ’A’);
    CHECK(list.size() == 24);
    CHECK(list.get(23) == ’B’);

    // remove something in the middle
    CHECK(list.remove(12) == ’M’);
    CHECK(list.size() == 23);
    CHECK(list.get(12) == ’L’);

    // check exception handling when access is beyond end of the list
    bool flag = true;
    try {
        list.remove(26);    // illegal access; element 26 doesn't exist
        flag = false;       // this line should not be reached due to exception
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

/*
 * Remove front element
 */
template <class T>
T DLL<T>::removeFirst() {
    if(pHead == 0) {
        throw std::out_of_range("Empty␣list␣in␣DLL::removeFirst()");
    }

    // save data in front node, and pointer to the node
    T d = pHead->data;
    Node *pTemp = pHead;

    // update head pointer
    pHead = pHead->pNext;
    if(pHead != 0) {
        // if there are more elements, mark new front node prev pointer
        // as left end of the list
        pHead->pPrev = 0;
    } else {
        // if there are no more elements, update tail pointer
        pTail = 0;
    }

    // update size, free former front node memory
    n--;
    delete pTemp;

    // send back value from former front node
    return d;
}

// doctest unit test for the removeFirst method
TEST_CASE("testing␣DLL<T>::removeFirst") {
    DLL<int> list;

    // populate list
    for(int i = 0; i < 10; i++) {
        list.addLast(i);
    }

    // check removing from front
    for(int i = 0; i < 10; i++) {
        CHECK(list.removeFirst() == i);
        CHECK(list.size() == 9 - i);
    }

    // check removing from empty list
    bool flag = true;
    try {
        list.removeFirst();     // list is empty, so this is an error
        flag = false;           // should never happen due to exception
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

/*
 * Remove back element.
 */
template <class T>
T DLL<T>::removeLast() {
    if(pHead == 0) {
        throw std::out_of_range("Empty␣list␣in␣DLL::removeLast()");
    }

    // save data in front node, and pointer to the node
    T d = pTail->data;
    Node *pTemp = pTail;

    // update head pointer
    pTail = pTail->pPrev;
    if(pTail != 0) {
        // if there are more elements, mark new last node next pointer
        // as right end of the list
        pTail->pNext = 0;
    } else {
        // if there are no more elements, update head pointer
        pHead = 0;
    }

    // update size, free former last node memory
    n--;
    delete pTemp;

    // send back value from former last node
    return d;
}

// doctest unit test for the removeLast method
TEST_CASE("testing␣DLL<T>::removeLast") {
    DLL<int> list;

    // populate list
    for(int i = 0; i < 10; i++) {
        list.addFirst(i);
    }

    // test removeLast
    for(int i = 0; i < 10; i++) {
        CHECK(list.removeLast() == i);
        CHECK(list.size() == 9 - i);
    }

    // test exception handling
    bool flag = true;
    try {
        list.removeLast();      // should not be legal; list is empty
        flag = false;           // should never happen due to exception
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

/*
 * Change the value at location idx to d.
 */
template <class T>
void DLL<T>::set(unsigned idx, const T &d) {
    // if the idx is past list end, throw an exception
    if(idx >= n) {
        throw std::out_of_range("Index␣out␣␣of␣range␣in␣DLL::set()");
    }

    // initialize cursor
    Node *pCurr = pHead;

    // iterate to location
    for(unsigned i = 0u; i < idx; i++) {
        pCurr = pCurr->pNext;
    }

    // change data in location idx to d
    pCurr->data = d;
}

// doctest unit test for the set method
TEST_CASE("testing␣DLL<T>::set") {
    DLL<char> list;

    // populate the list
    for(char c = 'A'; c <= 'Z'; c++) {
        list.addFirst(c);
    }

    // set first element
    list.set(0, ’z’);
    CHECK(list.get(0) == ’z’);

    // set last element
    list.set(25, ’a’);
    CHECK(list.get(25) == ’a’);

    // set something in the middle
    list.set(13, ’m’);
    CHECK(list.get(13) == ’m’);

    // check exception handling for index beyond end of list
    bool flag = true;
    try {
        list.set(26, ’X’);  // this is illegal; index doesn’t exist
        flag = false;       // this should never be reached, due to exception
    } catch(std::out_of_range oor) {
        CHECK(flag);    // if exception was handled properly, should be true
    }
}

/*
 * Change the value at the front to d.
 */
template <class T>
void DLL<T>::setFirst(const T &d) {
    // throw an exception if the list is empty
    if(isEmpty()) {
        throw std::out_of_range("Empty␣list␣in␣DLL<T>::setFirst()");
    }

    pHead->data = d;
}

// doctest unit test for setFirst
TEST_CASE("testing␣DLL<T>::setFirst") {
    DLL<char> list;

    // populate list
    for(char c = ’a’; c <= ’z’; c++) {
        list.addLast(c);
    }

    // test setFirst
    list.setFirst(’A’);
    CHECK(list.getFirst() == ’A’);
    list.clear();
    list.addFirst(’A’);
    list.setFirst(’a’);
    CHECK(list.getLast() == ’a’);

    // test exception handling
    bool flag = true;
    list.clear();
    try {
        list.setFirst(’Q’);     // should cause an exception
        flag = false;           // should never happen
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

/*
 * Change the value at the back to d.
 */
template <class T>
void DLL<T>::setLast(const T &d) {
    // throw an exception if the list is empty
    if(isEmpty()) {
        throw std::out_of_range("Empty␣list␣in␣DLL<T>::setLast()");
    }

    pTail->data = d;
}

// doctest unit test for setLast
TEST_CASE("testing␣DLL<T>::setLast") {
    DLL<char> list;

    // populate list
    for(char c = ’a’; c <= ’z’; c++) {
        list.addLast(c);
    }

    // test setFirst
    list.setLast(’A’);
    CHECK(list.getLast() == ’A’);
    list.clear();
    list.addFirst(’A’);
    list.setLast(’a’);
    CHECK(list.getFirst() == ’a’);

    // test exception handling
    bool flag = true;
    list.clear();
    try {
        list.setLast(’Q’);     // should cause an exception
        flag = false;           // should never happen
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

/*
 * Assignment operator.
 */
template <class T>
DLL<T> & DLL<T>::operator=(const DLL<T> &otherList) {
    // remove any existing contents first
    clear();

    // copy other list contents to this object
    copy(otherList);

    return *this;
}

// doctest unit test for the assignment operator
TEST_CASE("testing␣DLL<T>␣assignment") {
    DLL<int> list1, list2;

    // populate lists
    for(int i = 0; i < 5; i++) {
        list1.addFirst(i);
        if(i % 2 == 0) {
            list2.addFirst(i);
        }
    }

    // do the assignment
    list1 = list2;

    // right size?
    CHECK(list1.size() == list2.size());

    // same contents?
    for(unsigned i = 0; i < list1.size(); i++) {
        CHECK(list1.get(i) == list2.get(i));
    }
}

/*
 * Override of the stream insertion operator. Using iterators removes
 * the need for this to be a friend of the DLL class.
 */
template <class T>
std::ostream &operator<<(std::ostream &out, const DLL<T> &list) {

    out << "[";

    // iterate through the list using an iterator
    typename DLL<T>::Iterator i = list.front();

    while(i != list.end()) {

        out << *i;

        // output comma for all but last element
        i++;
        if(i != list.end()) {
            out << ",␣";
        }
    }

    out << "]";

    return out;
}

// doctest unit test for the stream insertion operator
TEST_CASE("testing␣DLL<T>␣stream␣insertion") {
    DLL<int> list;

    for(int i = 0; i < 5; i++) {
        list.addFirst(i);
    }

    // test stream insertion by "printing" to a string
    std::ostringstream oss;

    oss << list;

    // did the output match?
    CHECK(oss.str() == "[4,␣3,␣2,␣1,␣0]");
}
```

Listing 2: Stack.hpp

```
#pragma once

#include <doctest.h>
#include <iostream>
#include <stdexcept>
#include <sstream>
#include "../1-DLL/DLL.hpp"

/*---------------------------------------------------------------------------
 * class definition
 *---------------------------------------------------------------------------*/

/**
 * @brief CMP 246 Module 5 generic stack.
 *
 * Stack is a generic stack data structure. It supports push, pop, and peek
 * methods, and clear, isEmpty, and size administrative methods. It also has
 * a copy constructor, and overrides the assignment and stream insertion
 * operators.
 */
template <class T> class Stack {
public:
    /**
     * @brief Default constructor.
     *
     * Make a new, empty stack.
     */
    Stack() {}

    /**
     * @brief Copy constructor.
     *
     * Make a new stack, just like the parameter.
     *
     * @param stack Constant reference to the stack to copy from.
     */
    Stack(const Stack<T> &stack) { copy(stack); }

    /**
     * @brief Empty the stack.
     *
     * Removes all elements from this stack.
     */
    void clear() { list.clear(); }

    /**
     * @brief Determine if the stack is empty.
     *
     * @return True if the stack is empty, false otherwise.
     */
    bool isEmpty() { return list.isEmpty(); }

    /**
     * @brief Access top element.
     *
     * @return The top element of the stack, without
     * removing the element from the stack.
     *
     * @throws std::out_of_range if the stack is empty.
     */
    T peek() const;

    /**
     * @brief Pop top element.
     *
     * Removes and returns the top element of the stack.
     *
     * @return Element of type T that was at the top of the stack.
     *
     * @throws std::out_of_range if the stack is empty.
     */
    T pop();

    /**
     * @brief Push a new element.
     *
     * Pushes a value of type T onto the top of the stack.
     *
     * @param v Element to push.
     */
    void push(const T &v) { list.addFirst(v); }

    /**
     * @brief Get the size of the stack.
     *
     * @return Number of elements in the stack.
     */
    unsigned size() { return list.size(); }

    /**
     * @brief Assignment operator override.
     *
     * @param stack Stack to copy from.
     *
     * @return Reference to this stack, after it has copied the parameter,
     * for chaining purposes.
     */
    Stack<T> &operator=(const Stack<T> &stack);

    /**
     * @brief Stream insertion override.
     *
     * @param out ostream object to output to, e.g., cout.
     *
     * @param stack Stack to output.
     *
     * @return Reference to the out ostream object.
     */
    friend std::ostream &operator<<(std::ostream &out, const Stack<T> &stack) {
        out << stack.list;
        return out;
    }
```

```

private:
    /**
     * Doubly-linked list used as the underlying data store for the stack.
     */
    DLL<T> list;

    /**
     * @brief Copy stack contents.
     *
     * Private helper method for copy constructor and assignment override.
     *
     * @param stack Stack to copy from.
     */
    void copy(const Stack<T> &stack) { list = stack.list; }
};

//-----------------------------------------------------------------------------
// function implementations
//-----------------------------------------------------------------------------

// doctest unit test for the copy constructor
TEST_CASE("testing␣Stack<T>::Stack(stack)␣constructor") {
    Stack<char> source;

    for(char c = ’a’; c <= ’z’; c++) {
        source.push(c);
    }

    Stack<char> stackCopy(source);

    // should have same elements, in the same order
    while(!stackCopy.isEmpty()) {
        char a = source.pop();
        char b = stackCopy.pop();
        CHECK(a == b);
    }

    // make sure both are empty
    bool res = source.isEmpty() && stackCopy.isEmpty();
    CHECK(res);
}

// doctest unit test for the clear method
TEST_CASE("testing␣Stack<T>::clear()") {
    Stack<int> stack;

    for(int i = 0; i < 10; i++) {
        stack.push(i);
    }

    // clear and valdiate size is zero
    stack.clear();

    CHECK(stack.size() == 0u);
    CHECK(stack.isEmpty());
}

// doctest unit test for the isEmpty method
TEST_CASE("testing␣Stack<T>::isEmpty()") {
    Stack<double> stack;

    // initial one is empty?
    CHECK(stack.isEmpty());

    // populate
    for(int i = 0; i < 10; i++) {
        stack.push(i);
    }

    // should not be empty
    CHECK(!stack.isEmpty());

    // clear
    stack.clear();

    // should be empty again
    CHECK(stack.isEmpty());
}

// copy is private, and so it is tested in copy constructor, assignment tests

/*
 * Peek function implementation.
 */
template <class T>
T Stack<T>::peek() const {
    if(list.isEmpty()) {
        throw std::out_of_range("Empty␣stack␣in␣Stack::peek()");
    }

    return list.getFirst();
}

// doctest unit test for the peek method
TEST_CASE("testing␣Stack<T>::peek()") {
    Stack<char> stack;

    for(char c = ’a’; c <= ’z’; c++) {
        stack.push(c);
    }

    // check access to top element
    char c = stack.peek();

    CHECK(c == ’z’);

    stack.pop();

    c = stack.peek();
    CHECK(c == ’y’);

    // check exception handling
    stack.clear();
    bool flag = true;
    try {
        c = stack.peek();   // should cause an exception
        flag = false;       // should never happen
    } catch(std::out_of_range oor) {
        // flag should still be true
        CHECK(flag);
    }
}

/*
 * Pop function implementation.
 */
template <class T>
T Stack<T>::pop() {
    if(list.isEmpty()) {
        throw std::out_of_range("Empty␣stack␣in␣Stack::ppo()");
    }

    return list.removeFirst();
}

// doctest unit test for pop function
TEST_CASE("testing␣Stack<T>::pop()") {
    Stack<int> stack;
    for(int i = 0; i < 10; i++) {
        stack.push(i);
    }

    // check pop always removes top element
    for(int i = 9; i >= 0; i--) {
        CHECK(stack.pop() == i);
        CHECK(stack.size() == 10 - (10 - i));
    }

    CHECK(stack.isEmpty());

    // check exception handling
    bool flag = true;
    try {
        stack.pop();    // should cause an exception
        flag = false;   // should never happen
    } catch (std::out_of_range oor) {
        // flag should still be true
        CHECK(flag);
    }
}

// doctest unit test for push function
TEST_CASE("testing␣Stack<T>::push()") {
    Stack<char> stack;
    stack.push(’a’);
    CHECK(stack.peek() == ’a’);
    CHECK(stack.size() == 1u);

    stack.push(’b’);
    CHECK(stack.size() == 2u);
    CHECK(stack.pop() == ’b’);
    CHECK(stack.peek() == ’a’);
}

// doctest unit test for size function
TEST_CASE("testing␣Stack<T>::size()") {
    Stack<int> stack;
    CHECK(stack.size() == 0);
    for(int i = 0; i < 10; i++) {
        stack.push(i);
        CHECK(stack.size() == (i + 1));
    }

    for(int i = 10; i >= 1; i--) {
        stack.pop();
        CHECK(stack.size() == (i - 1));
    }
}

/*
 * Assignment operator override.
 */
template <class T>
Stack<T> &Stack<T>::operator=(const Stack<T> &stack) {
    copy(stack);
    return *this;
}

// doctest unit test for the assignment operator
TEST_CASE("testing␣Stack<T>␣assignment␣operator") {
    Stack<char> source;

    for(char c = ’a’; c <= ’z’; c++) {
        source.push(c);
    }

    Stack<char> stackCopy = source;

    // should have same elements, in the same order
    while(!stackCopy.isEmpty()) {
        char a = source.pop();
        char b = stackCopy.pop();
        CHECK(a == b);
    }

    // make sure both are empty
    bool res = source.isEmpty() && stackCopy.isEmpty();
    CHECK(res);
}

// doctest unit test for the stream insertion operator
TEST_CASE("testing␣Stack<T>␣stream␣insertion") {
    Stack<int> stack;

    for(int i = 0; i < 5; i++) {
        stack.push(i);
    }

    // test stream insertion by "printing" to a string
    std::ostringstream oss;

    oss << stack;

    // did the output match?
    CHECK(oss.str() == "[4,␣3,␣2,␣1,␣0]");
}
```

Listing 3: Queue.hpp

```
#pragma once

#include <doctest.h>
#include <iostream>
#include <stdexcept>
#include <sstream>
#include <string>
#include "../1-DLL/DLL.hpp"

/*---------------------------------------------------------------------------
 * class definition
 *--------------------------------------------------------------------------*/

/**
 * @brief CMP 246 Module 5 generic queue.
 *
 * Queue is a generic queue data structure. It supports enqueue and dequeue
 * methods, and clear, isEmpty, and size administrative methods. It also has
 * a copy constructor, and overrides the assignment and stream insertion
 * operators.
 */
template <class T> class Queue {
public:
    /**
     * @brief Default constructor.
     *
     * Make a new, empty queue.
     */
    Queue() { }

    /**
     * @brief Copy constructor.
     *
     * Make a new queue, just like the parameter.
     *
     * @param queue Constant reference to the queue to copy from.
     */
    Queue(const Queue<T> &queue) { copy(queue); }

    /**
     * @brief Empty the queue.
     *
     * Removes all elements from this queue.
     */
    void clear() { list.clear(); }

    /**
     * @brief Dequeue first element.
     *
     * Removes and returns the first element from the queue.
     *
     * @return Element of type T that was the first element in the queue.
     *
     * @throws std::out_of_range if the queue is empty
     */
    T dequeue();

    /**
     * @brief Enqueue a new element.
     *
     * Adds a value of type T to the back of the queue.
     *
     * @param v Element to enqueue
     */
    void enqueue(const T& v) { list.addLast(v); }

    /**
     * @brief Determine if the queue is empty.
     *
     * @return True if the queue is empty, false otherwise.
     */
    bool isEmpty() { return list.isEmpty(); }

    /**
     * @brief Assignment operator override.
     *
     * @param queue Queue to copy from.
     *
     * @return Reference to this queue, after it has copied the parameter,
     * for chaining purposes.
     */
    Queue<T> &operator=(const Queue<T> &queue);

    /**
     * @brief Stream insertion override.
     *
     * @param out ostream object to output to, e.g., cout.
     *
     * @param queue Queue to output.
     *
     * @return Reference to the out ostream object.
     */
    friend std::ostream &operator<<(std::ostream &out, const Queue<T> &queue) {
        out << queue.list;
        return out;
    }

    /**
     * @brief Get the size of the queue.
     *
     * @return Number of elements in the queue.
     */
    unsigned size() { return list.size(); }

private:
    /**
     * Doubly-linked list used as the underlying data store for the queue.
     */
    DLL<T> list;

    /**
     * @brief Copy queue contents.
     *
     * Private helper method for copy constructor and assignment override.
     *
     * @param queue Queue to copy from.
     */
    void copy(const Queue<T> &queue) { list = queue.list; }
};

//----------------------------------------------------------------------------
// function implementations
//----------------------------------------------------------------------------

// doctest unit tests for copy constructor
TEST_CASE("Testing␣Queue<T>::Queue(queue)␣constructor") {
    Queue<int> q1;
    for(int i = 1; i <= 5; i++) {
        q1.enqueue(i);
    }

    Queue<int> q2(q1);

    for(int i = 1; i <= 5; i++) {
        CHECK(i == q2.dequeue());
    }
}

// doctest unit tests for clear method
TEST_CASE("Testing␣Queue<T>::clear()") {
    Queue<char> q;
    for(char c = ’a’; c <= ’z’; c++) {
        q.enqueue(c);
    }
    q.clear();
    CHECK(q.isEmpty());
}

// copy is private, so it is tested in copy constructor and assignment
// operator tests

/*
 * Dequeue method.
 */
template <class T>
T Queue<T>::dequeue() {
    if(list.isEmpty()) {
        throw std::out_of_range("dequeue␣of␣empty␣queue");
    }

    return list.removeFirst();
}

// doctest unit tests for dequeue method
TEST_CASE("Testing␣Queue<T>::dequeue()") {
    Queue<int> q;
    for(int i = 0; i < 5; i++) {
        q.enqueue(i);
    }

    for(int i = 0; i < 5; i++) {
        CHECK(i == q.dequeue());
        CHECK((4 - i) == q.size());
    }

    bool flag = true;
    try {
        q.dequeue();  // should cause an exception
        flag = false;   // should never happen
    } catch(std::out_of_range oor) {
        CHECK(flag);
    }
}

// doctest unit tests for enqueue method
TEST_CASE("Testing␣Queue<T>::enqueue()") {
    Queue<double> q;

    for(int i = 1; i <= 5; i++) {
        q.enqueue(i);
        CHECK(i == q.size());
    }

    for(int i = 1; i <= 5; i++) {
        CHECK(i == q.dequeue());
    }
}

// doctest unit tests for isEmpty method
TEST_CASE("Testing␣Queue<T>::isEmpty()") {
    Queue<std::string> q;

    CHECK(q.isEmpty());

    q.enqueue("Bob");

    CHECK(!q.isEmpty());

    q.dequeue();

    CHECK(q.isEmpty());
}

/*
 * Implementation for assignment operator.
 */
```

```
template <class T>
Queue<T> & Queue<T>::operator=(const Queue<T> &queue) {
    copy(queue);
    return *this;
}

// doctest unit test for assignment operator
TEST_CASE("Testing␣Queue<T>::operator=") {
    Queue<int> q1, q2;

    for(int i = 0; i < 5; i++) {
        q1.enqueue(i);
        q2.enqueue(5 - i);
    }
    q2.enqueue(-1);

    q2 = q1;

    CHECK(5 == q2.size());
    for(int i = 0; i < 5; i++) {
        CHECK(i == q2.dequeue());
    }
}

// doctest unit test for the stream insertion operator
TEST_CASE("testing␣Queue<T>␣stream␣insertion") {
    Queue<int> q;

    for(int i = 0; i < 5; i++) {
        q.enqueue(i);
    }

    // test stream insertion by "printing" to a string
    std::ostringstream oss;

    oss << q;

    // did the output match?
    CHECK(oss.str() == "[0,␣1,␣2,␣3,␣4]");
}

// doctest unit tests for size method
TEST_CASE("Testing␣Queue<T>::size") {
    Queue<char> q;

    CHECK(0 == q.size());

    for(int i = 1; i <= 5; i++) {
        q.enqueue(’a’);
        CHECK(i == q.size());
    }

    for(int i = 4; i >= 0; i--) {
        q.dequeue();
        CHECK(i == q.size());
    }
}
```

Listing 4: RPN.cpp

```
#include <cstdlib>
#include <iostream>
#include <set>
#include <string>
#include "../2-Stack/Stack.hpp"

/**
 * Main program for the Doane CMP 246 Module 5 RPN calculator.
 */
int main() {

    // welcome prompt
    std::cout << "Welcome␣to␣the␣Doane␣RPN␣Calculator!" << std::endl;
    std::cout << "Please␣enter␣an␣expression␣in␣postfix,␣EOF␣to␣quit."
        << std::endl;

    // prepare stack
    Stack<double> stack;

    // read string tokens until there is nothing more to read
    std::string token;
    double a, b;
    while(std::cin >> token) {
        if(token == "E") {
            // print result of expression
            std::cout << ">>␣" << stack.pop() << std::endl;
            stack.clear();
        } else if(token == "+") {
            // addition
            b = stack.pop();
            a = stack.pop();
            stack.push(a + b);
        } else if(token == "-") {
            // subtraction
            b = stack.pop();
            a = stack.pop();
            stack.push(a - b);
        } else if(token == "*") {
            // multiplication
            b = stack.pop();
            a = stack.pop();
            stack.push(a * b);
        } else if(token == "/") {
            // division
            b = stack.pop();
            a = stack.pop();
            stack.push(a / b);
        } else {
            // numeric entry
            stack.push(atof(token.c_str()));
        }
    }

    // good bye prompt
    std::cout << "Good␣bye!" << std::endl;

    return EXIT_SUCCESS;
}
```