CS 11 Data Structures and Algorithms

Assignment 10: Linked Lists 2

Return to Course Homepage

Assignment 10.1

```
#ifndef SEQUENCE_H
#define SEQUENCE_H

namespace cs_sequence {

    class sequence {
    public:
        typedef std::size_t size_type;
        typedef int value_type;
        sequence();
        sequence(const sequence& source);
        ~sequence();
        sequence& operator=(const sequence& source);
        size_type size() const;
        void start();
        value_type current() const;
        void advance();
        bool is_item() const;
        void insert(const value_type& entry);
        void attach(const value_type& entry);
        void remove_current();
    private:
        struct node {
            value_type data;
            node* next;
        };
        node* headptr;
        node* tailptr;
        node* cursor;
        node* precursor;
        size_type numitems;
        void copy(const sequence& source);
        void clear();
    };




    // PRIVATE (HELPER) MEMBER FUNCTIONS FIRST

#include <cassert>

    void sequence::copy(const sequence& source){
        numitems = source.numitems;
        precursor = NULL;
        cursor = NULL;

        if (source.headptr == NULL) {
            headptr = NULL;
            tailptr = NULL;
        } else {
            headptr = new node;
            headptr -> data = source.headptr -> data;
            headptr -> next = NULL;
            node* sourceptr = source.headptr -> next;
            node* curptr = headptr;
            if (source.headptr == source.cursor){
                cursor = curptr;
            }
            while (sourceptr != NULL) {
                curptr -> next = new node;
                if (sourceptr == source.cursor) {
                    cursor = curptr -> next;
                    precursor = curptr;
                }
                curptr = curptr -> next;
                curptr -> data = sourceptr -> data;
                curptr -> next = NULL;
                sourceptr = sourceptr -> next;
            }
            tailptr = curptr;
        }
    }




    void sequence::clear() {
        if (headptr != NULL) {
            node* delptr = headptr;
            while (delptr != NULL) {
                headptr = headptr->next;
                delete delptr;
                delptr = headptr;
            }
        }
    }




    // NOW PUBLIC MEMBER FUNCTIONS, STARTING WITH THE BIG 4

    sequence::sequence()
    {
        numitems = 0;
        headptr = NULL;
        tailptr = NULL;
        cursor = NULL;
        precursor = NULL;
    }




    sequence::sequence(const sequence& source) {
        copy(source);
    }




    sequence::~sequence(){
        clear();
    }




    sequence& sequence::operator=(const sequence& source) {
        if (this != &source) {
            clear();
            copy(source);
        }
        return *this;
    }




    sequence::size_type sequence::size() const {
        return numitems;
    }




    void sequence::start() {
        cursor = headptr;
        precursor = NULL;
    }
```

```
sequence::value_type sequence::current() const {
    assert(is_item());
    return cursor -> data;
}
```

```
void sequence::advance() {
    assert(is_item());
    precursor = cursor;
    cursor = cursor -> next;
    if (cursor == NULL) {
        precursor = NULL;
    }
}
```

```
bool sequence::is_item() const {
    return cursor != NULL;
}
```

```
void sequence::insert(const value_type& entry) {

    node* new_node = new node;
    new_node->data = entry;
    numitems++;

    if (cursor == headptr || cursor == nullptr) { // insert at front (or into empty list).
        new_node->next = headptr;                 // precursor remains nullptr.
        headptr = new_node;
        if (numitems == 1) {
            tailptr = new_node;
        }
    } else {                                       // inserting anywhere else
        new_node->next = cursor;                   // tailptr, headptr and precursor don't change.
        precursor->next = new_node;
    }

    cursor = new_node;
}
```

```
void sequence::attach(const value_type& entry) {
    numitems++;
    node* tempptr = new node;
    tempptr -> data = entry;


    if (headptr == NULL) {                                  // attaching onto empty list.
        tempptr -> next = NULL;                             // precursor remains NULL.
        headptr = tempptr;
        tailptr = tempptr;
    } else if (cursor == NULL || cursor == tailptr) {       // attaching at end.
        tempptr -> next = NULL;
        tailptr -> next = tempptr;
        precursor = tailptr;
        tailptr = tempptr;
    } else {                                                // attaching anywhere else.
        tempptr -> next = cursor -> next;
        cursor -> next = tempptr;
        precursor = cursor;
    }

    cursor = tempptr;
}
```

```
    void sequence::remove_current() {
        assert(is_item());
        numitems--;

        if (headptr == tailptr) {
            delete headptr;
            headptr = NULL;
            tailptr = NULL;
            cursor = NULL;
        } else if (cursor == headptr) {
            headptr = cursor -> next;
            delete cursor;
            cursor = headptr;
        } else {
            node* tempptr = cursor;
            precursor -> next = cursor -> next;
            cursor = cursor -> next;
            if (cursor == NULL) {
                tailptr = precursor;
                precursor = NULL;
            }
            delete tempptr;
        }
    }
}
#endif




/*
 Here are some alternate solutions for insert and attach.

void sequence::insert(const value_type& entry) {
    numitems++;
    node* tempptr = new node;
    tempptr -> data = entry;

    if (cursor == NULL) {
        cursor = headptr;               // so entry will be inserted at front when cursor is NULL.
    }

    tempptr -> next = cursor;           // connect the new node to the node that will come after it
    // (might be NULL).
    if (headptr == NULL) {
        tailptr = tempptr;              // if the list is empty, need to set tailptr to the new node.
    }

    if (cursor == headptr) {
        headptr = tempptr;              // if inserting at front, set headptr.  precursor remains NULL.
    } else {
        precursor -> next = tempptr;    // if inserting anwhere else, connect precursor to the new node.
    }

    cursor = tempptr;                   // cursor will always point at the node just inserted.
}




void sequence::attach(const value_type& entry) {

    numitems++;
    node* tempptr = new node;
    tempptr -> data = entry;

    if (cursor == NULL) {
        cursor = tailptr;
    }

    if (cursor == tailptr) {
        tailptr = tempptr;
    }

    if (headptr == NULL) {
        tempptr -> next = cursor;
        headptr = tempptr;
    } else {
        tempptr -> next = cursor -> next;
        cursor -> next = tempptr;
        precursor = cursor;
    }

    cursor = tempptr;
}
*/
```