Joanna Liu  
UID: 405-957-891

Spring 2023 CS32 - Project 2 Report

a. My doubly linked list is implemented so that it is circular and has a dummy node. Each node in the list contains the data members m\_data, m\_next, and m\_prev, where m\_next and m\_prev are Node pointers that point to the next and previous nodes respectively, and m\_data being of type ItemType, used for storing some value in the sequence. The doubly linked list has a Node pointer head that points to the dummy node in the linked list to indicate the beginning/end of the Sequence. The dummy node’s m\_prev pointer points to the last node in the Sequence, and the dummy node’s m\_next pointer points to the first node in the Sequence. The dummy node’s m\_data variable is undefined and never accessed. Below is a picture of what the doubly linked list looks like when the list is empty and when it has been filled with a few values.



b. Below is the pseudocode for the functions in my Sequence class that have algorithms that are considered to be non-trivial.

Sequence::remove   
*Traverse through the entire sequence, starting at the beginning.  
 For each item, if the item in the sequence is equal to the value passed in the parameter,  
 Erase that item from the sequence using erase().  
 Increment the number of times an item has been removed from the sequence.  
 Else, keep traversing the sequence until reaching the end.  
Return the number of times an item has been removed from the sequence.*

Sequence::erase   
*If the position that is being erased is not within the bounds of the sequence,   
 Return false.  
Otherwise,   
 Traverse the array until you reach the node in the position passed through the parameter.  
 Once the position has been reached,   
 Move the previous node’s next pointer to point to the node after the node to be deleted.  
 Move the next node’s previous pointer to point to the node before the node to be deleted.  
 Delete the node that is supposed to be deleted.  
 Decrement the size of the sequence.  
 Return true.*

Sequence::insert, two arguments  
*If the position passed in the parameter is out of range,  
 Return -1.  
Traverse the sequence until you are one index before the position passed as a parameter, call this node k.  
Make another pointer keeping track of the node after the position you are currently at, call this node p.  
Create a new node.  
Fill the node with the value passed as a parameter.  
Assign the new node’s next pointer to the node p.  
Assign the node k’s next pointer to the new node.  
Assign the new node’s previous pointer to the node k.  
Assign the node p’s previous pointer to the new node.  
Increment the size of the sequence.  
Return the position passed as a parameter.*

Sequence::insert, one argument:  
*Traverse each item in the sequence:  
 If the item isn’t larger than the value passed in the parameter,  
 Keep traversing through the sequence.  
 Otherwise,  
 Use the insert() with two values to insert the value into the position we are currently at.  
 Return the index we inserted at.  
Insert the value at the end of the sequence and return the return value of insert().*

Sequence::subsequence   
*If either of the sequences are empty, or if the first sequence is smaller than the second,  
 Return false;  
Set the return value to -1 to signify we haven’t found a match.  
Set the checkpoint to the beginning of sequence 1.  
Repeatedly traverse through seq1 from the checkpoint:  
 If the item in seq1 is equal to seq2’s first item,  
 Store what position their first items matched in the return value and checkpoint.  
 Repeatedly continue traversing seq1, now also traversing seq2 from the beginning:  
 If the value in seq1 is not equal to the value in seq2,  
 Set the return value back to -1.  
 Break.  
 If seq2 was fully traversed without breaking,  
 Break.  
Return the return value.*

Sequence::concatReverse  
*Make a new sequence to store the result in.  
Repeat until you’ve copied over every value in seq1:  
 Get values, starting from the end of seq1 and going to the beginning.  
 Copy the values into the end of the new sequence.  
Repeat until you’ve copied over every value in seq2:  
 Get values, starting from the end of seq2 and going to the beginning.  
 Copy the values into the end of the new sequence.  
Swap the sequence passed as a parameter (result) with the new sequence.  
Delete the new sequence (which is now storing the old result sequence).*

c. The following code is the file that I used to test my Sequence class. The tests were performed on a sequence of strings, with ItemType as a type alias for std::string. The explanation of the purpose of each of my tests is written as a comment in green text.

#include "Sequence.h"

#include <iostream>

#include <cassert>

using namespace std;

void test()

{

Sequence s;

ItemType getter;

// Testing basic functions for an empty sequence

assert(s.size() == 0);

assert(s.empty());

assert(s.remove("hello") == 0);

assert(s.erase(10) == false);

assert(s.set(0, "goodbye") == false);

assert(s.get(10, getter) == false);

assert(s.find("hello") == -1);

// Adding items to the sequence

assert(s.insert("cheese") == 0); // checking insert

assert(s.insert("potato") == 1); // checking that insert without index inserts items in order

assert(s.insert("potato") == 1); // checking repeats of words it still inserts correctly

assert(s.insert(0, "apple") == 0); // checking insert with index

assert(s.insert(3, "banana") == 3); // checking insert at m\_size adds item to the end

assert(s.insert(6, "tomato") == -1); // should not insert because outside of sequence

assert(s.size() == 5); // checks that size changed properly

assert(s.empty() == false); // checks that empty works properly

assert(s.get(2, getter) == true); // checking get() returns the right value when position is valid

assert(getter == "potato"); // checking get() gets the correct value

assert(s.get(6, getter) == false); // checking get() returns correct value when position is not valid

assert(getter == "potato"); // checking getter isn't changed when position is out of the sequence

assert(s.find("banana") == 3); // checking find() returns correctly when in sequence

assert(s.find("tomato") == -1); // checking find() returns correctly when not in sequence

// Removing items from the sequence

assert(s.erase(4) == true); // checking erase when there are two of the same value

assert(s.erase(2) == true); // checking erase when position is in bounds

assert(s.erase(6) == false); // checking erase when position is not in bounds

assert(s.insert(0, "banana") == 0); // adding a duplicate string to list to test remove

assert(s.remove("banana") == 2); // checking that remove() counts the removed items correctly

assert(s.size() == 2); // making sure the bananas were removed from the sequence

// Testing swap():

Sequence t;

t.swap(s); // swapping with an empty sequence

assert(t.size() == 2); // making sure size is changed

assert(s.size() == 0); // making sure size is changed

// Making sure content in sequences changed:

cerr << "Expected t: " << "apple cheese" << endl << "Actual t: ";

t.dump();

cerr << "Expected s: " << "" << endl << "Actual s: ";

s.dump();

// refilling up s

for (int i = 0; i < 5; i++) {

s.insert(i, "bread");

}

t.swap(s); // swapping two filled lists

cerr << "Expected t: " << "bread bread bread bread bread" << endl << "Actual t: ";

t.dump();

cerr << "Expected s: " << "apple cheese" << endl << "Actual s: ";

s.dump();

t.swap(t); // swapping with itself

cerr << "Expected t: " << "bread bread bread bread bread" << endl << "Actual t: ";

t.dump();

// Testing copy constructor

Sequence x = t; // copying a filled sequence

assert(x.size() == t.size()); // checking size was copied correctly

cerr << "Expected x: " << "bread bread bread bread bread" << endl << "Actual x: ";

x.dump();

Sequence y;

Sequence z = y; // copying from an empty sequence

assert(z.empty()); // checking that the copied sequence is empty as well

cerr << "Expected z: " << "" << endl << "Actual z: ";

z.dump();

// Testing assignment operator

x = y; // testing with empty sequence (x should be empty)

assert(x.empty());

x = t; // testing with a filled sequence

assert(x.size() == 5);

// Test concatReverse()

t.remove("bread"); // emptying old sequences so they can be used for new tests

// adding more values into the sequences so they can be tested:

s.insert(0, "pear"); // s = pear, apple, cheese

t.insert("potato");

t.insert("tomato");

t.insert("yogurt");

t.insert("butter"); // t = butter potato tomato yogurt

concatReverse(s, t, y); // concatReverse() into an empty sequence

assert(y.size() == 7);

concatReverse(t, s, x); // concatReverse() into a filled sequence

assert(x.size() == 7);

cerr << "Expected y: " << "cheese apple pear yogurt tomato potato butter" << endl << "Actual y: ";

y.dump();

cerr << "Expected x: " << "yogurt tomato potato butter cheese apple pear" << endl << "Actual x: ";

x.dump();

concatReverse(s, s, s); // concatReverse() into itself

cerr << "Expected s: " << "cheese apple pear cheese apple pear" << endl << "Actual s: ";

s.dump();

// Test subsequence()

s.erase(3);

s.erase(3);

s.erase(3); // s = cheese apple pear

z.insert("bread");

z.insert("omelette"); // z = bread omelette

assert(subsequence(s, y) == -1); // when seq2 > seq1

assert(subsequence(x, s) == 4); // when seq2 IS in seq1

assert(subsequence(y, z) == -1); // when seq2 ISNT in seq1

y.insert(0, "cheese");

assert(subsequence(y, s) == 1); // when seq2 is in seq1 but not at the first place you find seq2\_0

Sequence u;

assert(subsequence(y, u) == -1); // when one of the sequences is empty

}

int main()

{

test();

cout << "Passed all tests" << endl;

}

When the above code is built with Sequence.h and Sequence.cpp, the following output is created:  
  
Expected t: apple cheese  
Actual t: apple cheese

Expected s:  
Actual s:

Expected t: bread bread bread bread bread  
Actual t: bread bread bread bread bread

Expected s: apple cheese  
Actual s: apple cheese

Expected t: bread bread bread bread bread  
Actual t: bread bread bread bread bread

Expected x: bread bread bread bread bread  
Actual x: bread bread bread bread bread

Expected z:  
Actual z:

Expected y: cheese apple pear yogurt tomato potato butter  
Actual y: cheese apple pear yogurt tomato potato butter

Expected x: yogurt tomato potato butter cheese apple pear  
Actual x: yogurt tomato potato butter cheese apple pear

Expected s: cheese apple pear cheese apple pear  
Actual s: cheese apple pear cheese apple pear

Passed all tests