Sagar Doshi

4/21/18

Dis 1A

TA: Abani, N.

CS 32 Project 2 Report

Design Description

I implemented the sequence class using a doubly linked list with a head and tail. My linked list did NOT have a dummy node, and it was NOT circular. The sequence class had the member variables head and tail to keep track of the head and tail, and a member variable size to keep track of the size of the sequence. The other member of the sequence class was an inner class called Node where each Node had a value, a pointer to the next Node, and a pointer to the previous node. Visually, the design was:

“A”

“B”

“C”

“D”

“E”

head

tail

size = 5

next

prev

value

null

null

Pseudocode for non-trivial functions

**Sequence()**

*set head and tail to null, and size to 0*

**Sequence(const Sequence &copy)**

*Check if copy is empty, if so just set head and tail to null and size to 0*

*If not empty, make a deep copy by:*

*Creating a temporary Node pointer to link between dynamically allocated nodes*

*Iterating through the size of copy…*

*Create a new node with value equal to copy’s value at that “index”*

*Use temp to store last node allocated, and use this to set next and prev*

*Don’t forget to set head to first allocated node and tail to second allocated node*

*Don’t forget special cases for head and tail where prev/next = nullptr*

*Don’t forget to copy over size*

**Sequence& operator=(const Sequence & copy);**

*Basically the same as copy constructor except return \*this and*

*Delete previously used memory by iterating backwards until head is reach and then delete head*

**~Sequence();**

*Start at tail and iterate backwards deleting allocated memory*

*After loop, delete last node located at head*

**Empty() and size() are trivial functions (same implementation as homework 1)**

**bool insert(int pos, const ItemType& value);**

*Return false if position out of range of list*

*Consider cases of adding on the end/start and empty list in addition to middle (care w/nullptr)*

*Use a finder pointer to iterate to the position*

*Allocate memory for new node and set value, next, and prev appropriately*

*Don’t forget to change next and previous of adjacent nodes*

*return true (successful insert)*

**int Sequence::insert(const ItemType & value)**

*This function can never return -1*

*Consider cases of adding on end/start and empty list in addition to middle (care w/nullptr)*

*Use a finder pointer to iterate through sequence keeping track of position with integer*

*if value at value <= finder->value*

*store position*

*call the insert(position, value) function with the position tracked*

*return position*

**bool Sequence::erase(int pos)**

*if Sequence is empty or position is out of bounds return false*

*use a finder pointer to iterate through sequence, keep track of position*

*once position is reached “bridge the gap” of adjacent nodes*

*(careful at start/end with nullptr), don’t forget to move head/tail*

*by appropriate changing next and prev*

*delete the node at position after bridging the gap*

**int Sequence::remove(const ItemType & value)**

*Return 0 if sequence is empty*

*Use a counter to keep track of amount of nodes deleted*

*Iterate through Sequence w/a finder pointer*

*If finder->value == value\_to\_be\_removed*

*Using the same protocol as erase, considering erasing the first and last nodes*

*Delete and bridge the gap, update the removal counter*

*Move on to the next node, and once u reach the end return the removal counter*

***bool Sequence::get(int pos, ItemType & value) const***

*Iterate through Sequence with a finder pointer until the position is reached*

*Consider empty case, first node, and last node cases (quick access w/head and tail)*

*Access finder->value at this position and set passed value to this value*

**bool Sequence::set(int pos, const ItemType & value)**

*Do the same thing as get but instead…*

*Set finder-> value to value passed.*

**void Sequence::swap(Sequence & other)**

*Very easy if copy and assignment operator are implemented correctly*

*Create a temp Node = other;*

*Other = this*

*This = temp*

**void Sequence::dump() const**

*Print out size of sequence*

*iterate through sequence with finder pointer and print out “Index: value”*

**int subsequence(const Sequence & seq1, const Sequence & seq2)**

*Check for empty case/if seq2.size() > seq1.size and return -1 if either of these cases happen*

*Create a starter value using get(0,start) on seq2 – use this for comparing values of seq1*

*Iterate through seq1 up until seq2 woudn’t be able to fit in seq 1, and keep track of the index of for loop, look for match*

*If a match is found -> enter an inner for loop that does a separate iteration*

*Iterate through both seq2 and seq1 until end of seq2, if entirety of seq2*

*Is contained with seq1, return the position of the index, otherwise continue iterating*

*through sequence*

*return the position in seq1 where subsequence started*

**void interleave(const Sequence & seq1, const Sequence & seq2, Sequence & result)**

*Essential to have working copy constructor and assignment operator*

*Consider alias cases of either seq1 or seq2 = result, and case where all three are the same*

*In case of aliases, be careful not to delete the nodes you want to copy over w/assignment*

*Operator*

*Consider cases of sizes being equal, or seq1 < seq2 and seq1>seq2 in terms of size.*

*Iterate through the larger sequence alternating the insertion of seq1 and seq2 into result*

*Note that a counter for position will be needed separate from for loop (care w/indeces)*

*Iterate through the runoff of larger sequence and append it to result*

*Be sure that when adding to result (excluding alias cases) result is empty*

*In cases of one of the lists being empty, use copy constructor/assignment operator to save*

*Runtime.*

Test Cases - were on string (ItemType == string) – dump to check ordering for certain functions where just assert is not suffcicient – no memory leaks on g++ either

/////////////////////Testing size(), empty(), dump(), insert(pos, value)///////////////////// - PASSED

Sequence s;

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

assert(s.empty());

//Entering into an empty list

assert(s.insert(0, "A") == true);

assert(s.size() == 1);

assert(s.empty() == false);

//Entering into the 0th index of a non-empty list

assert(s.insert(0, "first") == true);

//Entering onto the end of a non-empty list

assert(s.insert(2, "last") == true);

//Entering into the middle/2nd element

assert(s.insert(1, "middle") == true);

assert(s.insert(2, "middle2") == true);

//Entering in the 2nd to last element

assert(s.insert(s.size() - 1, "2nd to last") == true);

//testing position too high or (-)

assert(s.insert(s.size() + 1, "nope") == false);

assert(s.insert(-1, "nope") == false);

s.insert(s.size(), "last2");

//s.dump();

// ///////////////Test insert(value)////////////////////////////// - PASSED

Sequence a;

//Test adding to an empty list

assert(a.empty() == true && a.size() == 0);

assert(a.insert("A") == 0);

assert(a.empty() == false && a.size() == 1);

//Test adding onto the end of a non-empty list

assert(a.insert("B") == 1);

//assert(a.insert("B") == 1); adding onto the 2nd element works as well

assert(a.insert("C") == 2);

assert(a.insert("D") == 3);

//Adding onto the 0th element of a non-zero list

assert(a.insert("A") == 0);

//Adding an element equal to the last element of the list

assert(a.insert("D") == 4);

//Adding in the middle

assert(a.insert("C")== 3);

assert(a.insert("C") == 3);

assert(a.insert("C") == 3);

assert(a.insert("C") == 3);

assert(a.insert("B") == 2);

// ///////////////////////////Test erase(pos) ////////////////////////////////////// - PASSED

Sequence b;

assert(b.empty() == true && b.size() == 0);

//Try to remove from an empty list

assert(b.erase(0) == false);

//Remove the element from the empty leset

b.insert("A");

assert(b.erase(0) == true);

b.insert("A");

b.insert("B");

b.insert("C");

b.insert("D");

b.insert("E");

//b.dump();

//Try to remove with a position out of range (-) or too high

assert(b.erase(b.size()) == false);

assert(b.erase(-1) == false);

//Removing first element in a list with size() > 1

assert(b.erase(0) == true);

b.insert("A");

//Removing 2nd element

assert(b.erase(1) == true);

b.insert("B");

//Removing 3rd element

assert(b.erase(2) == true);

b.insert("C");

//Removing 2nd to last element

assert(b.erase(3) == true);

b.insert("D");

//removing last element

assert(b.erase(4) == true);

////////////////////////////Test remove(value);///////////////////////////// - PASSED

Sequence c;

///////////Testing removal of singl elements/////////////////////// - PASSED

//Removing from an empty list returns 0

assert(c.remove("anything") == 0);

c.insert("A");

c.insert("B");

c.insert("C");

c.insert("D");

c.insert("E");

//Removing first element

assert(c.remove("A") == 1);

c.insert("A");

//Removing 2nd element

assert(c.remove("B") == 1);

c.insert("B");

//Removing 3rd Element

assert(c.remove("C") == 1);

c.insert("C");

//Removing 4th element

assert(c.remove("D") == 1);

c.insert("D");

//Removing last element

assert(c.remove("E") == 1);

c.insert("E");

///////////////////////Testing of removal of elements/////////////////////////////// - PASSED

//Removing Multiple of the first element

c.insert("A");

c.insert("A");

assert(c.remove("A") == 3);

assert(c.size() == 4);

c.insert("A");

//Removing multiple elements in the middle (including 2nd and 2nd to last element)

Sequence c1;

c1.insert("A");

c1.insert("B");

c1.insert("B");

c1.insert("B");

c1.insert("C");

assert(c1.remove("B") == 3);

//Removing multiple elements at the end

c.insert("E");

c.insert("E");

assert(c.remove("E") == 3);

// /////////////////////////TESTING get() //////////////////////// - PASSED

Sequence g;

ItemType empty;

//Check if getting from an empty list fails

assert(g.get(0, empty) == false);

g.insert("A");

g.insert("B");

g.insert("C");

g.insert("D");

g.insert("E");

ItemType s1;

ItemType s2;

ItemType s3;

ItemType s4;

ItemType s5;

//Check if too high of a pos or (-) pos fails

assert(g.get(-1, s1) == false);

assert(g.get(6, s1) == false);

//Check all positions to see if get() works

assert(g.get(0, s1) == true && s1 == "A");

assert(g.get(1, s2) == true && s2 == "B");

assert(g.get(2, s3) == true && s3 == "C");

assert(g.get(3, s4) == true && s4 == "D");

assert(g.get(4, s5) == true && s5 == "E");

//g.dump();

// /////////////////////TESTING set() //////////////////////////// - PASSED

Sequence h;

//Check if trying to set somethign in an empty list fails

assert(h.set(0, "empty") == false);

h.insert("A");

h.insert("B");

h.insert("C");

h.insert("D");

h.insert("E");

//h.dump();

//Check if out of bounds of (-) pos fails

assert(h.set(-1, "false") == false);

assert(h.set(100, "false") == false);

assert(h.set(0, "E") == true);

assert(h.set(1, "D") == true);

assert(h.set(2, "C") == true);

assert(h.set(3, "B") == true);

assert(h.set(4, "A") == true);

//Check if the Itemtype can be reversed

//h.dump();

//

// //////////////Testing finder() ////////////////////////// - PASSED

//

//Check if trying to find somethign in an empty Itemtype is always -1

assert(empty.find("anything?") == -1);

assert(empty.find("not there") == -1);

//Make sure all found values have the correct position

assert(h.find("E") == 0);

assert(h.find("D") == 1);

assert(h.find("C") == 2);

assert(h.find("B") == 3);

assert(h.find("A") == 4);

//Make sure if unfound value inputted -1 is returned

assert(h.find("NOPE") == -1);

//h.dump();

/////////////////////////////////Testing of Copy CONSTRUCTOR////////////////////////////////////////////// - PASSED

Sequence empty\_sequence;

Sequence x;

x.insert("A");

//Testing empty list case

Sequence only\_1 = empty\_sequence;

assert(only\_1.size() == 0 && only\_1.empty() == true);

//Testing case 0f 1 element copy

Sequence only\_1\_2 = x;

assert(only\_1\_2.size() == 1);

x.insert("B");

//Testing case of only 2 eleemnt copy

Sequence only\_2 = x;

assert(only\_2.size() == 2);

x.insert("C");

x.insert("D");

x.insert("E");

//Testing case of copying the entire thing

Sequence copy = x;

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

//copy.dump();

///////////////////////////////Testing of assigment operator////////////////////////////////// - PASSED

Sequence assignment\_1;

Sequence assignment\_2;

assignment\_2.insert("A");

//Testing empty list case

assignment\_2 = empty\_sequence;

assert(assignment\_2.size() == 0 && assignment\_2.empty() == true);

//Testing case 0 f 1 element copy

assignment\_1.insert("A");

assignment\_2 = assignment\_1;

assert(assignment\_2.size() == 1);

//assignment\_2.dump();

assignment\_1.insert("B");

//Testing case of only 2 element copy into an empty list

assignment\_2 = assignment\_1;

assert(assignment\_2.size() == 2);

//assignment\_2.dump();

assignment\_1.insert("C");

assignment\_1.insert("D");

assignment\_1.insert("E");

//Testing case of copying the entire thing into an empty list

assignment\_2 = assignment\_1;

assert(assignment\_2.size() == 5);

//assignment\_2.dump();

//test of aliasing

assignment\_1 = assignment\_2;

assignment\_1.insert("A");

//Check of copying a non empty list into a non-empty list

assignment\_2 = assignment\_1;

//assignment\_2.dump();

///////////////////////////Testing swap (uses copy constructor and assignment operator)/////////////////// - PASSED

Sequence x1;

Sequence x2;

//Case of swapping empty sequences

x1.swap(x2);

assert(x1.size() == 0 && x1.empty() == true);

assert(x2.size() == 0 && x2.empty() == true);

x2.swap(x1);

assert(x1.size() == 0 && x1.empty() == true);

assert(x2.size() == 0 && x2.empty() == true);

//Swapping sequence with only element with an empty sequence

x1.insert("x1");

x1.swap(x2);

assert(x1.size() == 0 && x1.empty() == true);

assert(x2.size() == 1 && x2.empty() == false);

x2.swap(x1);

assert(x1.size() == 1 && x1.empty() == false);

assert(x2.size() == 0 && x2.empty() == true);

//Swapping a full size sequence with an empty sequence

x1.erase(0);

x1.insert("A");

x1.insert("B");

x1.insert("C");

x1.insert("D");

x1.insert("E");

x1.swap(x2);

assert(x1.size() == 0 && x1.empty() == true);

assert(x2.size() == 5 && x2.empty() == false);

x2.swap(x1);

assert(x1.size() == 5 && x1.empty() == false);

assert(x2.size() == 0 && x2.empty() == true);

//Swapping a 1 element sequence with a full size sequences.

x2.insert("x2");

x1.swap(x2);

assert(x1.size() == 1 && x1.empty() == false);

assert(x2.size() == 5 && x2.empty() == false);

x2.swap(x1);

assert(x1.size() == 5 && x1.empty() == false);

assert(x2.size() == 1 && x2.empty() == false);

//Swapping two full size sequences of same sizes

x2.erase(0);

x2.insert("1");

x2.insert("2");

x2.insert("3");

x2.insert("4");

x2.insert("5");

x1.swap(x2);

assert(x1.size() == 5 && x1.empty() == false);

assert(x2.size() == 5 && x2.empty() == false);

x2.swap(x1);

assert(x1.size() == 5 && x1.empty() == false);

assert(x2.size() == 5 && x2.empty() == false);

//Swapping two full size sequences of different sizes

x1.insert("F");

x1.swap(x2);

assert(x1.size() == 5 && x1.empty() == false);

assert(x2.size() == 6 && x2.empty() == false);

/\*x1.dump();

x2.dump();\*/

x2.swap(x1);

assert(x1.size() == 6 && x1.empty() == false);

assert(x2.size() == 5 && x2.empty() == false);

//x1.dump();

//x2.dump();

/////////////////////////Testing of interleave()////////////////////////////////// - PASSED

Sequence a1;

Sequence a2;

Sequence result;

assert(a1.empty() == true);

assert(a2.empty() == true);

assert(result.empty() == true);

//Don't set result equal to empty from here on out so we can check if something is wrong

//Interleaving empty strings

interleave(a1, a2, result);

//result.dump();

interleave(a2, a1, result);

//result.dump();

//interleaving an empty string with a non empty string (only 1 element)

a1.insert("A");

interleave(a1, a2, result);

//result.dump();

interleave(a2, a1, result);

//result.dump();

//Interleaving an empty string with a non empty string (2 elements)

a1.insert("B");

interleave(a1, a2, result);

//result.dump();

interleave(a2, a1, result);

//result.dump();

//Interleaving an empty string with a non empty string (5 elements)

a1.insert("C");

a1.insert("D");

a1.insert("E");

interleave(a1, a2, result);

//result.dump();

interleave(a2, a1, result);

//result.dump();

//Interleaving an non-empty string with a non empty string both 1 element

a1 = empty\_sequence;

a1.insert("A");

a2.insert("B");

//result = empty\_sequence;

interleave(a1, a2, result);

//result.dump();

interleave(a2, a1, result);;

//result.dump();

//Interleaving a non-empty string wtih a non empty string both 2 elements

a2 = empty\_sequence;

a1.insert("B");

a2.insert("1");

a2.insert("2");

interleave(a1, a2, result);

//result.dump();

interleave(a2, a1, result);

//result.dump();

//Interleaving a non-empty string wtih a non empty string both multiple elements

a1.insert("C");

a1.insert("D");

a1.insert("E");

a2.insert("3");

a2.insert("4");

a2.insert("5");

interleave(a1, a2, result);

//result.dump();

interleave(a2, a1, result);

//result.dump();

////////////////////Alias checking///////////////////// - PASSED

//Should NOT be empty

interleave(result, result, result);

//result.dump();

//Should not be empty

interleave(result, a2, result);

//result.dump();

//Should not be empty

interleave(a1, result, result);

//result.dump();

//In all of these cases, not an empty string, and result is added onto (checked with dump)

//Meaning that the memorh handling was correct

/////////////////////////////////Testing subseqeunce ///////////////////////

Sequence b1;

Sequence b2;

//Testing if both are empty

assert(subsequence(b1, b2) == -1);

//Testing if either of them are empty

b1.insert("A");

assert(subsequence(b1, b2) == -1);

assert(subsequence(b2, b1) == -1);

//Testing size of 1 (start)

b2.insert("A");

assert(subsequence(b1, b2) == 0);

b2 = empty\_sequence;

b2.insert("B");

assert(subsequence(b1, b2) == -1);

//Testing size of 2 (start)

b1.insert("B");

b2.insert("A");

//AB = AB

assert(subsequence(b1, b2) == 0);

b2 = empty\_sequence;

b2.insert("B");

//Order incorrect

b2.insert(1,"A");

assert(subsequence(b1, b2) == -1);

//Testing size of 5

b1 = empty\_sequence;

b2 = empty\_sequence;

b1.insert("A");

b1.insert("B");

b1.insert("C");

b1.insert("D");

b1.insert("E");

b2.insert("A");

assert(subsequence(b1, b2) == 0);

b2.erase(0);

b2.insert("B");

assert(subsequence(b1, b2) == 1);

b2.erase(0);

b2.insert("C");

assert(subsequence(b1, b2) == 2);

b2.erase(0);

b2.insert("D");

assert(subsequence(b1, b2) == 3);

b2.erase(0);

b2.insert("E");

assert(subsequence(b1, b2) == 4);

//Checking out of order

b2.erase(0);

b2.insert("BA");

assert(subsequence(b1, b2) == -1);

b2 = empty\_sequence;

b2.insert("A");

b2.insert("B");

assert(subsequence(b1, b2) == 0);

b2 = empty\_sequence;

b2.insert("B");

b2.insert(1, "C");

assert(subsequence(b1, b2) == 1);

b2 = empty\_sequence;

b2.insert("C");

b2.insert(1, "D");

assert(subsequence(b1, b2) == 2);

b2 = empty\_sequence;

b2.insert("D");

b2.insert(1, "E");

assert(subsequence(b1, b2) == 3);

b2 = empty\_sequence;

b2.insert("A");

b2.insert("B");

b2.insert("C");

assert(subsequence(b1, b2) == 0);

b2 = empty\_sequence;

b2.insert("A");

b2.insert("B");

b2.insert("C");

b2.insert("D");

assert(subsequence(b1, b2) == 0);

b2 = empty\_sequence;

b2.insert("A");

b2.insert("B");

b2.insert("C");

b2.insert("D");

b2.insert("E");

assert(subsequence(b1, b2) == 0);

b2 = empty\_sequence;

b2.insert("B");

b2.insert("C");

b2.insert("D");

assert(subsequence(b1, b2) == 1);

b2 = empty\_sequence;

b2.insert("C");

b2.insert("D");

b2.insert("E");

assert(subsequence(b1, b2) == 2);

b2 = empty\_sequence;

b2.insert("C");

b2.insert(1,"E");

b2.insert(2,"D");

//b2.dump();

//b1.dump();

assert(subsequence(b1, b2) == -1);

//Testing an out of order (should be -1)

////

////#include "Sequence.h"

////#include <type\_traits>

////

////#define CHECKTYPE(f, t) { auto p = static\_cast<t>(f); (void)p; }

////

////static\_assert(std::is\_default\_constructible<Sequence>::value,

//// "Sequence must be default-constructible.");

////static\_assert(std::is\_copy\_constructible<Sequence>::value,

//// "Sequence must be copy-constructible.");

////

////void thisFunctionWillNeverBeCalled()

////{

//// CHECKTYPE(&Sequence::operator=, Sequence& (Sequence::\*)(const Sequence&));

//// CHECKTYPE(&Sequence::empty, bool (Sequence::\*)() const);

//// CHECKTYPE(&Sequence::size, int (Sequence::\*)() const);

//// CHECKTYPE(&Sequence::insert, bool (Sequence::\*)(int, const ItemType&));

//// CHECKTYPE(&Sequence::insert, int (Sequence::\*)(const ItemType&));

//// CHECKTYPE(&Sequence::erase, bool (Sequence::\*)(int));

//// CHECKTYPE(&Sequence::remove, int (Sequence::\*)(const ItemType&));

//// CHECKTYPE(&Sequence::get, bool (Sequence::\*)(int, ItemType&) const);

//// CHECKTYPE(&Sequence::set, bool (Sequence::\*)(int, const ItemType&));

//// CHECKTYPE(&Sequence::find, int (Sequence::\*)(const ItemType&) const);

//// CHECKTYPE(&Sequence::swap, void (Sequence::\*)(Sequence&));

//// CHECKTYPE(subsequence, int(\*)(const Sequence&, const Sequence&));

//// CHECKTYPE(interleave, void(\*)(const Sequence&, const Sequence&, Sequence&));

////}

////

////int main()

////{}

//#include "Sequence.h"

//#include <string>

//#include <iostream>

//#include <cassert>

//using namespace std;

//

//void test()

//{

// Sequence s;

// assert(s.insert(0, "lavash"));

// assert(s.insert(0, "tortilla"));

// assert(s.size() == 2);

// ItemType x = "injera";

// assert(s.get(0, x) && x == "tortilla");

// assert(s.get(1, x) && x == "lavash");

//}

//

//int main()

//{

// test();

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

//}

//#include "Sequence.h"

//#include <iostream>

//#include <cassert>

//using namespace std;

//

//void test()

//{

// Sequence s;

// assert(s.insert(0, 10));

// assert(s.insert(0, 20));

// assert(s.size() == 2);

// ItemType x = 999;

// assert(s.get(0, x) && x == 20);

// assert(s.get(1, x) && x == 10);

//}

//

//int main()

//{

// test();

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

//}