Project 2 Report

Com Sci 32 – Spring 2019

Design Description

For the purpose of creating this project, circular doubly linked lists were utilized. This was done by creating a private struct Node within the class, where the data members were defined to be m\_data – an appropriate value of a given ItemType, and two pointers of type Node, namely m\_next and m\_prev to point to a particular Node objects successor and predecessor respectively. To avoid dealing with nullptr, a dummy node was also included in the list, the m\_data member variable of which was left uninitialized.

Here is a representation of an empty list according to my implementation:

head

???

Next is a representation of a typical linked list containing a finite number of items:

head

“Bye”

“Hi”

???

Pseudo-code for non-trivial functions

Following is the pseudo-code for the Sequence::remove() member function, which is somewhat non-trivial in the algorithm which it uses, which necessitates writing pseudo-code for it.

traverse through the linked list

if matching value is found

link predecessor and successor Nodes

delete present Node

list size --

number of deletions made ++

return number of deletions made

Next is the pseudo-code for the interleave() non-member function, the non-triviality of which also necessitates pseudo-code.

temp is temporary Sequence object

if both lists are of the same size

traverse through both sequences simultaneously

insert seq1’s present item into temp

insert seq2’s present item into temp

if seq1 is shorter than seq2

traverse through both sequences till seq1.size() items

insert seq1’s present item into temp

insert seq2’s present item into temp

traverse through remaining items in seq2

insert seq2’s present item into temp

if seq2 is shorter than seq1

traverse through both sequences till seq2.size() items

insert seq1’s present item into temp

insert seq2’s present item into temp

traverse through remaining items in seq1

insert seq1’s present item into temp

else if either list is empty

copy over the non-empty list to temp

copy temp over to result (prevents aliasing issues)

Lastly, we include the pseudo-code for the subsequence() non-member function, which is also non-trivial.

traverse through seq1

if an element matches with the first element of seq2

traverse through both sequences simultaneously to check if corresponding elements match

if any item doesn’t match set flag to false and continue traversing seq1

if the inner loop succeeds without running off either of the sequences

confirm that seq2 is a subsequence of seq1

else state that seq2 is not a subsequence of seq1

Test Cases

We divide our test cases between testing two datatypes – string and long. Within each datatype, the test cases explore various scenarios wherein each of the member/ non-member functions are tested thoroughly. For thoroughness, testing will employ two types of cases - where the member functions can perform legal operations appropriately, and where the member functions handle illegal operation requests appropriately.

Test Data for string (i.e. ItemType = std::string)

*// creating an empty Sequence*

Sequence s; *// default constructor creates an empty Sequence.*

assert(s.empty()); *// testing empty()*

assert(s.size()==0); *// testing size()*

assert(s.remove("Hello") == 0); *// nothing to remove.*

*// constructing a non-empty Sequence by successively inserting items into a previously empty Sequence.*

assert(s.insert(0, "apple") == 0); *// testing the first insert() function which uses a position and an ItemType value to be inserted. Successful insertion returns the position where insertion was made.*

assert(!s.empty()); *// s is non-empty now.*

assert(s.size()==1); *// s contains one item.*

assert(s.insert(1, "cat") == 1);

assert(s.size()==2); *// each time an item is inserted, size increases by one.*

assert(s.insert(2, "elephant") == 2);

assert(s.size()==3);

assert(s.insert(4, "fig") == -1); *// can't skip values while inserting. The position where insertion is to be made must lie between 0 and size().*

assert(s.size() == 3); *// unsuccessful insertion operation doesn't affect size.*

*// accessing data in a non-empty Sequence*

*// testing get() for each position in the Sequence.*

ItemType x = "";

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

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

assert(s.get(2, x) && x=="elephant");

assert(!s.get(3, x) && x=="elephant"); *// accessing outside bounds returns false and leaves x unaffected.*

*// testing find() for each ItemType value in the list.*

assert(s.find("elephant") == 2);

assert(s.find("cat") == 1);

assert(s.find("apple") == 0);

assert(s.find("zebra") == -1); *// cannot find what is not already there in the Sequence.*

*// modifying a non-empty Sequence in various ways*

assert(s.insert(1,"ball") == 1); *// inserting in the middle of a Sequence using the first insert function.*

assert(s.size() == 4); *// size of the sequence increases by one.*

assert(s.find("apple") == 0); *// items before the newly inserted item remain at the same position.*

*// all items after the inserted one are shifted ahead by one.*

assert(s.find("cat") == 2);

assert(s.find("elephant") == 3);

assert(s.insert("dog") == 3); *// testing the second insert() function, which inserts its argument wherever the arguments value (in this case lexicographical) is <= the value of the ItemType value at that position in the sequence. In this case, "dog"<="elephant", thus it will be inserted at the position where "elephant" was previously.*

assert(s.size() == 5); *// the size further increases by 1.*

*// items before "dog" remain at the same position as before.*

assert(s.find("apple") == 0);

assert(s.find("ball") == 1);

assert(s.find("cat") == 2);

*// "dog" was inserted where "elephant" was before due to reasons stated above.*

assert(s.find("dog") == 3);

assert(s.find("elephant") == 4); *// "elephant" was shifted one position ahead.*

assert(s.insert("fog") == 5); *// inserting something which is greater than all existing items in the sequence simply results in its insertion towards the end of the Sequence.*

assert(s.size() == 6);

assert(s.set(5, "freeway")); *// modifying a pre-existing element in the Sequence using set().*

assert(s.get(5, x) && x == "freeway"); *// checking whether or not the modification was made.*

assert(!s.set(10, "hallway")); *// cannot access out of bound positions.*

assert(s.erase(1)); *// successful deletion for an item within bounds. In this case, the item deleted is "ball".*

assert(s.size() == 5); *// size of the Sequence is reduced by 1 whenever erase() succeeds.*

assert(s.find("apple") == 0); *// items before the deleted item remain at the same position.*

*// items after the deleted item are shifted one position behind.*

assert(s.find("cat") == 1);

assert(s.find("dog") == 2);

assert(s.find("elephant") == 3);

assert(s.find("freeway") == 4);

assert(!s.erase(100)); *// cannot erase from out of bounds.*

assert(s.size() == 5); *// a failed erase() operation doesn't affect the size.*

assert(s.insert("freeway") == 4); *// it is okay for a Sequence to have two identical items. In this case both "freeway" and "freeway" have the same value, so the "freeway" that is already existing is shifted one up, and the new "freeway" is inserted in its place.*

assert(s.size() == 6);

assert(s.remove("freeway") == 2); *// calling remove() on "freeway" deletes both its instances from the Sequence.*

assert(s.size() == 4); *// as two items were deleted, size goes down by two.*

*// as both "freeway" instances were terminal elements for the Sequence, the items before them are retained at the same positions as before.*

assert(s.find("apple") == 0);

assert(s.find("cat") == 1);

assert(s.find("dog") == 2);

assert(s.find("elephant") == 3);

assert(s.remove("hallelujah") == 0); *// cannot remove an element that doesn't already exist in the Sequence.*

assert(s.size() == 4); *// a failed remove() call doesn't affect the size of the Sequence.*

Sequence s1(s); *// creating a new Sequence object from an already existing one. This is testing the copy constructor. s1 must have the same contents as s.*

assert(s1.size() == s.size()); *// s1 has the same size as s.*

*// the items at each position in s1 correspond to that of s.*

assert(s1.find("apple") == s.find("apple"));

assert(s1.find("cat") == s.find("cat"));

assert(s1.find("dog") == s.find("dog"));

assert(s1.find("elephant") == s.find("elephant"));

*// modifying s1, a copied over Sequence.*

assert(s1.set(0, "zebra"));

assert(s1.set(1, "xenophobia"));

assert(s1.set(2, "yttrium"));

assert(s1.set(3, "when"));

*// ensuring that changes were made to s1*

assert(s1.get(0, x) && x == "zebra");

assert(s1.get(1, x) && x == "xenophobia");

assert(s1.get(2, x) && x == "yttrium");

assert(s1.get(3, x) && x == "when");

*// any changes to s1 shouldn't affect s, as they are now independent entities.*

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

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

assert(s.get(2, x) && x == "dog");

assert(s.get(3, x) && x == "elephant");

*// swapping a Sequence with itself won't affect it.*

s.swap(s);

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

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

assert(s.get(2, x) && x == "dog");

assert(s.get(3, x) && x == "elephant");

s.swap(s1); *// swapping s and s1 with each other.*

*// ensuring that the swap worked on s.*

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

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

assert(s.get(2, x) && x == "yttrium");

assert(s.get(3, x) && x == "when");

*// ensuring that the swap worked on s1 as well.*

assert(s1.get(0, x) && x == "apple");

assert(s1.get(1, x) && x == "cat");

assert(s1.get(2, x) && x == "dog");

assert(s1.get(3, x) && x == "elephant");

*// creating a new Sequence object, and then overwriting it with s.*

Sequence s2;

assert(s2.insert(2, "hello") == -1); *// The first insertion index must be 0.*

assert(s2.insert(0, "parachute") == 0);

assert(s2.insert("quiet")); *// "quiet" > "parachute", so it will be appended at the end.*

assert(s2.insert("radiance")); *// "radiance" > "parachute" and "radiance" > "quiet" thus "radiance” is again appended at the end.*

s2 = s1; *// overwriting s2 with s1. This tests the assignment operator.*

*// updating contents of s2.*

assert(s2.set(0, "alpha"));

assert(s2.set(1, "beta"));

assert(s2.set(2, "gamma"));

assert(s2. set(3, "delta"));

*// changing s2 shouldn't affect s1, as they are independent entities.*

assert(s1.get(0, x) && x == "apple");

assert(s1.get(1, x) && x == "cat");

assert(s1.get(2, x) && x == "dog");

assert(s1.get(3, x) && x == "elephant");

*// testing various cases of the interleave function.*

Sequence s3; *// creating an empty Sequence to store the results of interleave.*

interleave(s1, s2, s3); *// overwriting an empty sequence with the interleave of s1 and s2.*

*// ensuring that s3 is properly formed.*

assert(s3.size() == 8);

assert(s3.get(0, x) && x == "apple");

assert(s3.get(1, x) && x == "alpha");

assert(s3.get(2, x) && x == "cat");

assert(s3.get(3, x) && x == "beta");

assert(s3.get(4, x) && x == "dog");

assert(s3.get(5, x) && x == "gamma");

assert(s3.get(6, x) && x == "elephant");

assert(s3.get(7, x) && x == "delta");

*// modifying s1 and s2*

*// deleting the last two elements out of both Sequences. This is done in two ways here.*

assert(s1.erase(2)); *// first deleting the second last element.*

assert(s1.erase(2)); *// then deleting the last element, shifted one index down.*

assert(s2.erase(3)); *// first deleting the last element.*

assert(s2.erase(2)); *// then deleting the second last element (which is now the last).*

*// ensuring that s1 and s2 are properly formed.*

assert(s1.get(0, x) && x == "apple");

assert(s1.get(1, x) && x == "cat");

assert(s2.set(0, "alpha"));

assert(s2.set(1, "beta"));

*// the last argument of interleave need not be an empty Sequence, as shown here (s is a pre-existing non-empty Sequence).*

interleave(s1, s2, s);

*// s should be appropriately overwritten from its previous value.*

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

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

assert(s.get(2, x) && x == "cat");

assert(s.get(3, x) && x == "beta");

*// in fact, this the implementation of interleave function makes it alias-proof, which means that arguments for the first and/ or second sequence can be same as the result sequence, as is seen in this case. Besides, in this case the size of s1 < s, so the ultimate state of s is as is specified by the spec.*

interleave(s, s1, s1);

*// s1 gets overwritten despite its previous value.*

assert(s1.size() == 6);

assert(s1.get(0, x) && x == "apple"); *// first element of s.*

assert(s1.get(1, x) && x == "apple"); *// (previously) first element of s1*

assert(s1.get(2, x) && x == "alpha"); *// second element of s*

assert(s1.get(3, x) && x == "cat"); *// (previously) second element of s1*

assert(s1.get(4, x) && x == "cat"); *// third element of s*

assert(s1.get(5, x) && x == "beta"); *// fourth element of s*

*// another evidence of interleave being alias-proof.*

interleave(s2, s2, s2);

*// s2 gets overwritten despite its previous value.*

assert(s2.size() == 4);

assert(s2.get(0, x) && x == "alpha");

assert(s2.get(1, x) && x == "alpha");

assert(s2.get(2, x) && x == "beta");

assert(s2.get(3, x) && x == "beta");

*// modifying s2*

assert(s2.erase(0)); *// deleting the first element of s2*

assert(s2.erase(2)); *// deleting the last element of s2*

assert(s2.size() == 2);

assert(s2.get(0, x) && x == "alpha");

assert(s2.get(1, x) && x == "beta");

*// we also consider a case where the size of the first argument of interleave is less than the first.*

interleave(s2, s, s1);

*// s1 gets overwritten despite its previous value.*

assert(s1.size() == 6);

assert(s1.get(0, x) && x == "alpha"); *// first element of s2*

assert(s1.get(1, x) && x == "apple"); *// first element of s*

assert(s1.get(2, x) && x == "beta"); *// second element of s2*

assert(s1.get(3, x) && x == "alpha"); *// second element of s*

assert(s1.get(4, x) && x == "cat"); *// third element of s*

assert(s1.get(5, x) && x == "beta"); *// fourth element of s*

Sequence s4; *// creating an empty Sequence to check some more cases of interleave.*

interleave(s4, s2, s1);

*// s1 is overwritten with the data of s2.*

assert(s1.size() == 2);

assert(s1.get(0, x) && x == "alpha");

assert(s1.get(1, x) && x == "beta");

interleave(s, s4, s1);

*// s1 is overwritten with the data of s*

assert(s1.get(0, x) && x == "apple");

assert(s1.get(1, x) && x == "alpha");

assert(s1.get(2, x) && x == "cat");

assert(s1.get(3, x) && x == "beta");

*// setting all other Sequences to empty Sequences as follows*

interleave(s4, s4, s);

assert(s.empty());

interleave(s4, s4, s1);

assert(s1.empty());

interleave(s4, s4, s2);

assert(s2.empty());

interleave(s4, s4, s3);

assert(s3.empty());

*// testing the subsequence function*

assert(subsequence(s1, s2) == -1); *// s2 can't be a subsequence of s1 if both are empty.*

assert(s1.insert("a1") == 0); *// s1 is now non-empty*

assert(subsequence(s1, s2) == -1); *// s2 is empty, so it isn't possible to find it in s1, which is a non-empty Sequence.*

assert(subsequence(s2, s1) == -1); *// s2 is empty, so it isn't possible to find s1, which is a non-empty Sequence, in it.*

*// updating seq1 and seq2*

assert(s1.insert(1, "a2") == 1);

assert(s1.insert(2, "a3") == 2);

assert(s1.insert(3, "a4") == 3);

assert(s2.insert(0, "a1") == 0);

assert(s2.insert(1, "a2") == 1);

assert(s2.insert(2, "b1") == 2);

*// every Sequence is a subsequence of itself.*

assert(subsequence(s1, s1) == 0);

assert(subsequence(s2, s2) == 0);

assert(subsequence(s1, s2) == -1); *// as s2 can't entirely be found in s1.*

assert(s2.erase(2)); *// deleting the last element of s2*

assert(subsequence(s1, s2) == 0); *// as s2 can now be found in s1, beginning at the 0th index.*

assert(s2.erase(0)); *// deleting first element of s2*

assert(s2.insert(1, "a3")); *// adding another element of the end of s2.*

assert(subsequence(s1, s2) == 1); *// s2 can be found as a subsequence of s1, beginning at the 1st index.*

*// updating s2.*

assert(s2.set(0, "a3"));

assert(s2.set(1, "a4"));

assert(s2.insert(2, "a5"));

assert(subsequence(s1, s2) == -1); *// although the first two elements of s2 were found in s1 as its last two elements, we can't go any further in s1 for checking whether or not s2 is a subsequence of s1, but some elements of s2 remain to be checked. So s2 cannot be concluded to be a subsequence of s1.*

assert(s2.erase(2));

assert(subsequence(s1, s2) == 2);

assert(subsequence(s2, s1) == -1); *// s1 is greater in size than s2, so clearly it can't be found as a subsequence of s2.*

assert(s2.remove("a4") == 1);

assert(s2.insert(0, "a1") == 0);

assert(subsequence(s1, s2) == -1); *// s2 cannot be found as a consecutive Sequence in s1, hence s2 cannot be considered to be a subsequence of s1.*

*// checking that only the first instance of the subsequence's starting position is returned.*

*// updating s2 for this purpose.*

assert(s2.insert(1, "a2") == 1);

assert(s2.insert(3, "a4") == 3);

assert(s2.insert(4, "b") == 4);

assert(s2.insert(5, "a1") == 5);

assert(s2.insert(6, "b1") == 6);

assert(s2.insert(7, "a1") == 7);

assert(s2.insert(8, "a2") == 8);

assert(s2.insert(9, "a3") == 9);

assert(s2.insert(10, "a4") == 10);

assert(s2.insert(11, "a5") == 11);

assert(subsequence(s2, s1) == 0); *// although s1 is found two times in s2, only its first occurrence in s2 is considered to give an output.*

*// in fact, even if s1 is consecutively present in s2, subsequence should still consider only its first occurrence.*

assert(s2.erase(4)); *// erasing the "b" in the middle of s2*

assert(s2.erase(4)); *// erasing the "a1" after "b"*

assert(s2.erase(4)); *// erasing the "b1" after "a1"*

assert(subsequence(s2, s1) == 0); *// although s1 occurs two times and consecutively so in s2, for subsequence, only the first occurrence is considered.*

*// the fact that g32 exits the program without any memory leak errors is a testimony to the validity of the destructor.*

Test Data for unsigned long (i.e. ItemType = unsigned long)

*// creating an empty Sequence*

Sequence s; *// default constructor creates an empty Sequence.*

assert(s.empty()); *// testing empty()*

assert(s.size()==0); *// testing size()*

assert(s.remove(12) == 0); *// nothing to remove.*

*// constructing a non-empty Sequence by successively inserting items into a previously empty Sequence.*

assert(s.insert(0, 10) == 0); *// testing the first insert() function which uses a position and an ItemType value to be inserted. Successful insertion returns the position where insertion was made.*

assert(!s.empty()); *// s is non-empty now.*

assert(s.size()==1); *// s contains one item.*

assert(s.insert(1, 12) == 1);

assert(s.size()==2); *// each time an item is inserted, size increases by one.*

assert(s.insert(2, 14) == 2);

assert(s.size()==3);

assert(s.insert(4, 20) == -1); *// can't skip values while inserting. The position where insertion is to be made must lie between 0 and size().*

assert(s.size() == 3); *// unsuccessful insertion operation doesn't affect size.*

*// accessing data in a non-empty Sequence*

*// testing get() for each position in the Sequence.*

ItemType x = 999;

assert(s.get(0, x) && x==10);

assert(s.get(1, x) && x==12);

assert(s.get(2, x) && x==14);

assert(!s.get(3, x) && x==14); *// accessing outside bounds returns false and leaves x unaffected.*

*// testing find() for each ItemType value in the list.*

assert(s.find(12) == 1);

assert(s.find(14) == 2);

assert(s.find(10) == 0);

assert(s.find(98) == -1); *// cannot find what is not already there in the Sequence.*

*// modifying a non-empty Sequence in various ways*

assert(s.insert(1,11) == 1); *// inserting in the middle of a Sequence using the first insert() function*

assert(s.size() == 4); *// size of the sequence increases by one.*

assert(s.find(10) == 0); *// items before the newly inserted item remain at the same position.*

*// all items after the inserted one are shifted ahead by one.*

assert(s.find(12) == 2);

assert(s.find(14) == 3);

assert(s.insert(13) == 3); *// testing the second insert() function, which inserts its argument wherever the arguments value is <= the value of the ItemType value at that position in the sequence. In this case, 13<=14, thus it will be inserted at the position where 14 was previously.*

assert(s.size() == 5); *// the size further increases by 1.*

*// items before 13 remain at the same position as before.*

assert(s.find(10) == 0);

assert(s.find(11) == 1);

assert(s.find(12) == 2);

*// 13 was inserted where 14 was before due to reasons stated above.*

assert(s.find(13) == 3);

assert(s.find(14) == 4); *// 14 was shifted one position ahead.*

assert(s.insert(15) == 5); *// inserting something which is greater than all existing items in the sequence simply results in its insertion towards the end of the Sequence.*

assert(s.size() == 6);

assert(s.set(5, 25)); *// modifying a pre-existing element in the Sequence using set().*

assert(s.get(5, x) && x == 25); *// checking whether or not the modification was made.*

assert(!s.set(10, 100)); *// cannot access out of bound positions.*

assert(s.erase(1)); *// successful deletion for an item within bounds. In this case, the item deleted is 11.*

assert(s.size() == 5); *// size of the Sequence is reduced by 1 whenever erase() succeeds.*

assert(s.find(10) == 0); *// items before the deleted item remain at the same position.*

*// items after the deleted item are shifted one position behind.*

assert(s.find(12) == 1);

assert(s.find(13) == 2);

assert(s.find(14) == 3);

assert(s.find(25) == 4);

assert(!s.erase(100)); *// cannot erase from out of bounds.*

assert(s.size() == 5); *// a failed erase() operation doesn't affect the size.*

assert(s.insert(25) == 4); *// it is okay for a Sequence to have two identical items. In this case both 25 and 25 have the same value, so the 25 that is already existing is shifted one up, and the new 25 is inserted in its place.*

assert(s.size() == 6);

assert(s.remove(25) == 2); *// calling remove() on 25 deletes both its instances from the Sequence.*

assert(s.size() == 4); *// as two items were deleted, size goes down by two.*

*// as both instances of 25 were terminal elements for the Sequence, the items before them are retained at the same positions as before.*

assert(s.find(10) == 0);

assert(s.find(12) == 1);

assert(s.find(13) == 2);

assert(s.find(14) == 3);

assert(s.remove(101) == 0); *// cannot remove an element that doesn't already exist in the Sequence.*

assert(s.size() == 4); *// a failed remove() call doesn't affect the size of the Sequence.*

Sequence s1(s); *// creating a new Sequence object from an already existing one. This is testing the copy constructor. s1 must have the same contents as s.*

assert(s1.size() == s.size()); *// s1 has the same size as s.*

*// the items at each position in s1 correspond to that of s.*

assert(s1.find(10) == s.find(10));

assert(s1.find(12) == s.find(12));

assert(s1.find(13) == s.find(13));

assert(s1.find(14) == s.find(14));

*// modifying s1, a copied over Sequence.*

assert(s1.set(0, 100));

assert(s1.set(1, 99));

assert(s1.set(2, 98));

assert(s1.set(3, 97));

*// ensuring that changes were made to s1*

assert(s1.get(0, x) && x == 100);

assert(s1.get(1, x) && x == 99);

assert(s1.get(2, x) && x == 98);

assert(s1.get(3, x) && x == 97);

*// any changes to s1 shouldn't affect s, as they are now independent entities.*

assert(s.get(0, x) && x == 10);

assert(s.get(1, x) && x == 12);

assert(s.get(2, x) && x == 13);

assert(s.get(3, x) && x == 14);

*// swapping a Sequence with itself won't affect it.*

s.swap(s);

assert(s.get(0, x) && x == 10);

assert(s.get(1, x) && x == 12);

assert(s.get(2, x) && x == 13);

assert(s.get(3, x) && x == 14);

s.swap(s1); *// swapping s and s1 with each other.*

*// ensuring that the swap worked on s.*

assert(s.get(0, x) && x == 100);

assert(s.get(1, x) && x == 99);

assert(s.get(2, x) && x == 98);

assert(s.get(3, x) && x == 97);

*// ensuring that the swap worked on s1 as well.*

assert(s1.get(0, x) && x == 10);

assert(s1.get(1, x) && x == 12);

assert(s1.get(2, x) && x == 13);

assert(s1.get(3, x) && x == 14);

*// creating a new Sequence object, and then overwriting it with s.*

Sequence s2;

assert(s2.insert(2, 120) == -1); *// The first insertion index must be 0.*

assert(s2.insert(0, 20) == 0);

assert(s2.insert(21)); *// 21 > 20, so it will be appended at the end.*

assert(s2.insert(22)); *// 22 > 21 and 22 > 20 thus 22 is again appended at the end.*

s2 = s1; *// overwriting s2 with s1. This tests the assignment operator.*

*// updating contents of s2.*

assert(s2.set(0, 110));

assert(s2.set(1, 112));

assert(s2.set(2, 113));

assert(s2. set(3, 114));

*// changing s2 shouldn't affect s1, as they are independent entities.*

assert(s1.get(0, x) && x == 10);

assert(s1.get(1, x) && x == 12);

assert(s1.get(2, x) && x == 13);

assert(s1.get(3, x) && x == 14);

*// testing various cases of the interleave function.*

Sequence s3; *// creating an empty Sequence to store the results of interleave.*

interleave(s1, s2, s3); *// overwriting an empty sequence with the interleave of s1 and s2.*

*// ensuring that s3 is properly formed.*

assert(s3.size() == 8);

assert(s3.get(0, x) && x == 10);

assert(s3.get(1, x) && x == 110);

assert(s3.get(2, x) && x == 12);

assert(s3.get(3, x) && x == 112);

assert(s3.get(4, x) && x == 13);

assert(s3.get(5, x) && x == 113);

assert(s3.get(6, x) && x == 14);

assert(s3.get(7, x) && x == 114);

*// modifying s1 and s2*

*// deleting the last two elements out of both Sequences. This is done in two ways here.*

assert(s1.erase(2)); *// first deleting the second last element.*

assert(s1.erase(2)); *// then deleting the last element, shifted one index down.*

assert(s2.erase(3)); *// first deleting the last element.*

assert(s2.erase(2)); *// then deleting the second last element (which is now the last).*

*// ensuring that s1 and s2 are properly formed.*

assert(s1.get(0, x) && x == 10);

assert(s1.get(1, x) && x == 12);

assert(s2.set(0, 110));

assert(s2.set(1, 112));

*// the last argument of interleave need not be an empty Sequence, as shown here (s is a pre-existing non-empty Sequence).*

interleave(s1, s2, s);

*// s should be appropriately overwritten from its previous value.*

assert(s.get(0, x) && x == 10);

assert(s.get(1, x) && x == 110);

assert(s.get(2, x) && x == 12);

assert(s.get(3, x) && x == 112);

*// in fact, this the implementation of interleave function makes it alias-proof, which means that arguments for the first and/ or second sequence can be same as the result sequence, as is seen in this case. Besides, in this case the size of s1 < s, so the ultimate state of s is as is specified by the spec.*

interleave(s, s1, s1);

*// s1 gets overwritten despite its previous value.*

assert(s1.size() == 6);

assert(s1.get(0, x) && x == 10); *// first element of s.*

assert(s1.get(1, x) && x == 10); *// (previously) first element of s1*

assert(s1.get(2, x) && x == 110); *// second element of s*

assert(s1.get(3, x) && x == 12); *// (previously) second element of s1*

assert(s1.get(4, x) && x == 12); *// third element of s*

assert(s1.get(5, x) && x == 112); *// fourth element of s*

*// another evidence of interleave being alias-proof.*

interleave(s2, s2, s2);

*// s2 gets overwritten despite its previous value.*

assert(s2.size() == 4);

assert(s2.get(0, x) && x == 110);

assert(s2.get(1, x) && x == 110);

assert(s2.get(2, x) && x == 112);

assert(s2.get(3, x) && x == 112);

*// modifying s2*

assert(s2.erase(0)); *// deleting the first element of s2*

assert(s2.erase(2)); *// deleting the last element of s2*

assert(s2.size() == 2);

assert(s2.get(0, x) && x == 110);

assert(s2.get(1, x) && x == 112);

*// we also consider a case where the size of the first argument of interleave is less than the first.*

interleave(s2, s, s1);

*// s1 gets overwritten despite its previous value.*

assert(s1.size() == 6);

assert(s1.get(0, x) && x == 110); *// first element of s2*

assert(s1.get(1, x) && x == 10); *// first element of s*

assert(s1.get(2, x) && x == 112); *// second element of s2*

assert(s1.get(3, x) && x == 110); *// second element of s*

assert(s1.get(4, x) && x == 12); *// third element of s*

assert(s1.get(5, x) && x == 112); *// fourth element of s*

Sequence s4; *// creating an empty Sequence to check some more cases of interleave.*

interleave(s4, s2, s1);

*// s1 is overwritten with the data of s2.*

assert(s1.size() == 2);

assert(s1.get(0, x) && x == 110);

assert(s1.get(1, x) && x == 112);

interleave(s, s4, s1);

*// s1 is overwritten with the data of s*

assert(s1.get(0, x) && x == 10);

assert(s1.get(1, x) && x == 110);

assert(s1.get(2, x) && x == 12);

assert(s1.get(3, x) && x == 112);

*// setting all other Sequences to empty Sequences as follows*

interleave(s4, s4, s);

assert(s.empty());

interleave(s4, s4, s1);

assert(s1.empty());

interleave(s4, s4, s2);

assert(s2.empty());

interleave(s4, s4, s3);

assert(s3.empty());

*// testing the subsequence function*

assert(subsequence(s1, s2) == -1); *// s2 can't be a subsequence of s1 if both are empty.*

assert(s1.insert(1) == 0); *// s1 is now non-empty*

assert(subsequence(s1, s2) == -1); *// s2 is empty, so it isn't possible to find it in s1, which is a non-empty Sequence.*

assert(subsequence(s2, s1) == -1); *// s2 is empty, so it isn't possible to find s1, which is a non-empty Sequence, in it.*

*// updating seq1 and seq2*

assert(s1.insert(1, 2) == 1);

assert(s1.insert(2, 3) == 2);

assert(s1.insert(3, 4) == 3);

assert(s2.insert(0, 1) == 0);

assert(s2.insert(1, 2) == 1);

assert(s2.insert(2, 11) == 2);

*// every Sequence is a subsequence of itself.*

assert(subsequence(s1, s1) == 0);

assert(subsequence(s2, s2) == 0);

assert(subsequence(s1, s2) == -1); *// as s2 can't entirely be found in s1.*

assert(s2.erase(2)); *// deleting the last element of s2.*

assert(subsequence(s1, s2) == 0); *// as s2 can now be found in s1, beginning at the 0th index.*

assert(s2.erase(0)); *// deleting first element of s2*

assert(s2.insert(1, 3)); *// adding another element of the end of s2.*

assert(subsequence(s1, s2) == 1); *// s2 can be found as a subsequence of s1, beginning at the 1st index.*

*// updating s2.*

assert(s2.set(0, 3));

assert(s2.set(1, 4));

assert(s2.insert(2, 5));

assert(subsequence(s1, s2) == -1); *// although the first two elements of s2 were found in s1 as its last two elements, we can't go any further in s1 for checking whether or not s2 is a subsequence of s1, but some elements of s2 remain to be checked. So s2 cannot be concluded to be a subsequence of s1.*

assert(s2.erase(2));

assert(subsequence(s1, s2) == 2);

assert(subsequence(s2, s1) == -1); *// s1 is greater in size than s2, so clearly it can't be found as a subsequence of s2.*

assert(s2.remove(4) == 1);

assert(s2.insert(0, 1) == 0);

assert(subsequence(s1, s2) == -1); *// s2 cannot be found as a consecutive Sequence in s1, hence s2 cannot be considered to be a subsequence of s1.*

*// checking that only the first instance of the subsequence's starting position is returned.*

*// updating s2 for this purpose.*

assert(s2.insert(1, 2) == 1);

assert(s2.insert(3, 4) == 3);

assert(s2.insert(4, 14) == 4);

assert(s2.insert(5, 0) == 5);

assert(s2.insert(6, 11) == 6);

assert(s2.insert(7, 1) == 7);

assert(s2.insert(8, 2) == 8);

assert(s2.insert(9, 3) == 9);

assert(s2.insert(10, 4) == 10);

assert(s2.insert(11, 19) == 11);

assert(subsequence(s2, s1) == 0); *// although s1 is found two times in s2, only its first occurrence in s2 is considered to give an output.*

*// in fact, even if s1 is consecutively present in s2, subsequence should still consider only its first occurrence.*

assert(s2.erase(4)); *// erasing the 14 in the middle of s2*

assert(s2.erase(4)); *// erasing the 0 after 14*

assert(s2.erase(4)); *// erasing the 11 after 0*

assert(subsequence(s2, s1) == 0); *// although s1 occurs two times and consecutively so in s2, for subsequence, only the first occurrence is considered.*

*// the fact that g32 exits the program without any memory leak errors is a testimony to the validity of the destructor.*