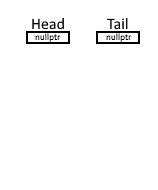
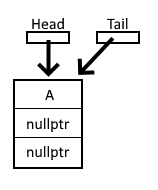
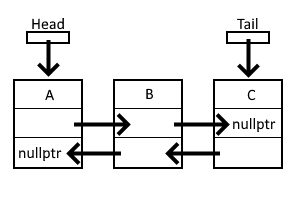
Project 2 report, by Tomas Kaljevic

My sequence uses a doubly-linked list implementation that contains three private data members: a pointer of type Node labeled head that points to the first node in the sequence, a pointer of type Node labeled tail that points to the last node in the sequence, and a data member of type int labeled size that holds the size of the sequence. If a sequence is empty, head and tail are null pointers. If a sequence contains one node, head and tail both point to that node. The following diagrams illustrate some typical sequences.



Pseudocode for non-trivial methods:

Sequence::~Sequence() {

Create temp node that points to head

For (i < num of nodes – 1)

Make head point to the node after it if it exists

Delete the node temp points to

Make temp point to the node head now points to

Delete the node head points to

}

Sequence::Sequence(const Sequence& other) {

Set the head and tail nodes to nullptr and the size to 0 (i.e. an empty sequence)

Insert nodes from other sequence to this sequence

}

Sequence& Sequence::operator=(Sequence rhs) {

If this sequence isn’t the sequence denoted by rhs

Create a temp sequence using the copy constructor that takes rhs as an argument

Swap the contents of this sequence with temp

}

int Sequence::insert(int pos, const ItemType& value) {

If the pos is out of bounds, return -1

Create a node to hold the value argument – call this node “item”

If pos is 0

If the sequence is empty, make head and tail point to item

Otherwise insert item before the head and make head point to it

Else if pos is the size of the sequence

Insert item after the tail and make tail point to it

Else

Create a “temp” pointer to the node before where we want to insert item

Make item’s next node point to temp’s next node

Make item’s previous node point to temp

Make temp’s next node point to item

Make the node after item’s previous node point to item

Increment the size of the sequence and return the pos

}

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

Loop through each node and find the first node with such that value <= that node’s value

Store the pos of that node (if it doesn’t exist, pos refers to the size of the sequence)

Return by making a function call to the other insert function

}

bool Sequence::erase(int pos) {

If the pos is out of bounds, return false

Create a “temp” pointer that points to the head

If pos is 0

If the sequence only has one element, set head and tail to nullptr

Otherwise make head point to the next node in the sequence

Delete the node temp points to

Else

Make temp point to the node before the one we want to delete

Create a “nodeToDelete” pointer that points to the node we want to delete

If the node after nodeToDelete isn’t null

Make temp’s next node point to nodeToDelete’s next node

Make the node after nodeToDelete’s previous node point to temp

Else

Make temp’s next node a nullptr

Make tail point to temp

Delete nodeToDelete

Decrement the size of the sequence and return true

}

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

If the sequence has only one element

Erase at pos=0 and return 1

Loop through each node and erase at pos=i if the node’s value is the value passed in

Return the number of nodes deleted

}

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

If the pos is out of bounds, return false

Get the node at the pos we want and make a “temp” pointer point to it

Assign the value argument to whatever the value of the node temp points to is

}

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

If the pos is out of bounds, return false

Get the node at the pos we want and make a “temp” pointer point to it

Assign the value of the node temp points to with the value argument

}

int Sequence::find(const ItemType& value) const {

Loop through each node and find a node with the value given

If it exists, return the pos it’s at

Return -1 otherwise

}

void Sequence::swap(Sequence& other) {

Swap the head pointer of this sequence with the other sequence

Swap the tail pointer of this sequence with the other sequence

Swap the size of this sequence with the other sequence

}

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

If the size of seq1 is less than the size of seq2 or if seq2 is empty

Return -1

For (i < size of seq1)

For (j < size of seq2)

If the value of the node at pos=(i + j) of seq1 is not the same as the value of the node at pos=j of seq2, there is no subsequence at pos=i in seq1

If they all match, the pos where there exists a subsequence is i

Return the pos at which there exists a subsequence or -1 if there isn’t one

}

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

If result isn’t empty, erase all the nodes in it

While we haven’t visited each element of seq1 and seq2

If there are more nodes in seq1

Add the value of that node to result at the next available position

If there are more nodes in seq2

Add the value of that node to result at the next available position

}

Program that includes a list of test cases to thoroughly test every function implemented:

Gist link (for readability because code isn’t formatted well in Word)

https://gist.github.com/tomaskalj/b603274a58f318ff0e4b6fb47b75d61c

Sequence s;  
{  
 assert(s.empty()); // test that the sequence is empty upon creation  
  
 assert(s.insert(3, "test1") == -1); // test that inserting a value greater than the sequence size is invalid  
  
 assert(s.insert(0, "a") != -1); // test that inserting values such that 0 <= pos < size is valid  
 assert(s.insert(1, "b") != -1);  
 assert(s.insert(2, "c") != -1);  
 assert(s.insert(3, "d") != -1);  
  
 assert(s.size() == 4); // test that the size function yields the correct size  
  
 assert(s.insert(s.size(), "e") != -1); // test that inserting values at pos = size is valid  
  
 assert(s.insert(0, "before head") != -1); // test that inserting a value before the head (i.e. pos = 0) is valid  
  
 assert(s.insert(2, "middle of sequence") != -1); // test that inserting a value in the middle of the sequence is valid  
}  
{  
 ItemType x;  
 assert(!s.get(10, x)); // test that trying to get the value at a position outside the size of the sequence returns false  
 assert(x.empty()); // show that x, since it is defined as a type of std::string, is empty as per the default std::string constructor  
  
 assert(s.get(3, x)); // test that getting the value at a pos such that 0 <= pos < size returns true  
 assert(x == "b"); // show that x is now whatever was at pos=3 of the sequence  
  
 assert(!s.set(10, "potato")); // test that trying to set the value at a position outside the size of the sequence returns false  
 assert(s.set(4, "e")); // test that setting the value at a pos such that 0 <= pos < size returns true  
  
 assert(s.find("AAAAA") == -1); // test that trying to find a value not in the sequence returns -1  
 assert(s.find("a") == 1); // test that trying to find a value in the sequence returns its correct position  
 assert(s.find("e") == 4); // test that trying to find a value in the sequence returns the first position it occurs at  
}  
{  
 ItemType y;  
 assert(!s.erase(10)); // test that trying to erase a node at a position greater than the size of the sequence returns false  
 assert(!s.erase(s.size())); // test that trying to erase a node at a position equal to the size of the sequence returns false  
 assert(s.erase(0)); // test that erasing the node at a pos such that 0 <= pos < size returns true  
  
 assert(s.get(0, y));  
 assert(y == "a"); // show that the value at the position we just deleted is the value of the node that was after the node we deleted  
  
 assert(s.remove("e") == 2); // test that removing a value returns the correct number of times that value was removed from the sequence  
  
 assert(s.size() == 4); // test that the sequence size was modified by removing N occurrences of "e"  
  
 for (int i = s.size() - 1; i >= 0; i--) {  
 assert(s.erase(i)); // test that removing every element within the bounds of the sequence works  
 }  
  
 assert(s.empty()); // test that the sequence is now empty  
}  
{  
 // general insertion tests  
 assert(s.insert(0, "a") != -1);  
 assert(s.insert(1, "b") != -1);  
 assert(s.insert(2, "b") != -1);  
 assert(s.insert(3, "c") != -1);  
 assert(s.insert(4, "c") != -1);  
 assert(s.insert(5, "c") != -1);  
 assert(s.insert(6, "d") != -1);  
 assert(s.insert(7, "d") != -1);  
 assert(s.insert(8, "d") != -1);  
 assert(s.insert(9, "d") != -1);  
  
 // general removal tests  
 assert(s.remove("d") == 4);  
 assert(s.remove("d") == 0); // test that removing a value that's not in the sequence returns 0  
  
 assert(s.remove("c") == 3);  
 assert(s.remove("c") == 0); // test that removing a value that's not in the sequence returns 0  
  
 assert(s.remove("b") == 2);  
 assert(s.remove("a") == 1);  
  
 assert(s.empty());  
}  
{  
 // test that one-argument insertion works  
 assert(s.insert("donkey") != -1);  
 assert(s.insert("apple") != -1);  
 assert(s.insert("elephant") != -1);  
 assert(s.insert("coconut") != -1);  
  
 // test that the items in the sequence are in the correct order  
 assert(s.find("apple") == 0);  
 assert(s.find("coconut") == 1);  
 assert(s.find("donkey") == 2);  
 assert(s.find("elephant") == 3);  
  
 // test that inserting a value inserts it in the correct position  
 assert(s.insert("banana") == 1);  
 assert(s.insert("d") == 3);  
}  
{  
 Sequence s2 = s; // set s3 to s using the assignment operator (which uses the copy constructor) to create a duplicate sequence  
  
 assert(s2.size() == 6); // test that s2 has the same size as s  
  
 assert(s2.erase(5)); // modify s2  
 assert(s2.erase(0));  
  
 assert(s2.size() == 4); // test that the size of s2 has been changed  
 assert(s.size() == 6); // test that the size of s has been unchanged  
  
 Sequence s3 = s2; // set s3 to s2 using the assignment operator  
  
 assert(s3.erase(0)); // modify s3  
  
 s.swap(s3); // swap the contents of s and s3  
 assert(s.size() == 3); // test that s now has the properties that s3 had  
 assert(s.find("coconut") == 0);  
 assert(s.find("d") == 1);  
 assert(s.find("donkey") == 2);  
  
 assert(s3.size() == 6); // test that s3 now has the properties that s had  
 assert(s3.find("coconut") == 2);  
 assert(s3.find("elephant") == 5);  
}  
{  
 Sequence seq1, seq2;  
 seq1.insert(0, "apple");  
 seq1.insert(1, "banana");  
 seq1.insert(2, "carrot");  
 seq1.insert(3, "donut");  
  
 seq2.insert(0, "apple");  
 seq2.insert("carrot");  
  
 assert(subsequence(seq1, seq2) == -1); // test that a subsequence does not exist for these two sequences  
  
 Sequence seq3;  
 assert(subsequence(seq1, seq3) == -1); // test that a subsequence does not exist if the second sequence passed in is empty  
  
 seq3.insert(0, "banana");  
 seq3.insert(1, "carrot");  
  
 assert(subsequence(seq1, seq3) == 1); // test that a subsequence exists for these two sequences  
  
 Sequence seq4, seq5;  
 seq4.insert("a");  
 seq5.insert("a");  
 seq5.insert("b");  
  
 assert(subsequence(seq4, seq5) == -1); // test that a subsequence cannot exist if the size of the first sequence  
 // passed in is less than the size of the second sequence passed in  
  
 Sequence seq6, seq7;  
 seq6.insert(0, "a");  
 seq6.insert(1, "b");  
 seq6.insert(2, "c");  
  
 seq7.insert(0, "a");  
 seq7.insert(1, "b");  
 seq7.insert(2, "c");  
  
 assert(subsequence(seq6, seq7) == 0); // test that the subsequence of two sequences that are identical is 0  
}  
{  
 Sequence seq1, seq2, result;  
 seq1.insert(0, "a");  
 seq1.insert(1, "c");  
 seq1.insert(2, "e");  
  
 seq2.insert(0, "b");  
 seq2.insert(1, "d");  
  
 interleave(seq1, seq2, result);  
 assert(result.size() == 5); // test that the properties of result are as they should be after an interleave operation  
 assert(result.find("b") == 1);  
 assert(result.find("d") == 3);  
  
 Sequence seq4;  
 interleave(seq2, seq4, result);  
 assert(result.size() == 2); // test that the the properties of result are as they should be if one of the sequences is empty  
 assert(result.find("b") == 0);  
 assert(result.find("d") == 1);  
  
 Sequence seq5;  
 interleave(seq4, seq5, result);  
 assert(result.empty()); // test that result is empty if both of the sequences given are empty  
  
 Sequence seq6, seq7;  
 seq6.insert(0, "axe");  
 seq6.insert(1, "zebra");  
  
 seq7.insert(0, "borat");  
  
 result = seq6; // make result equal to seq6  
  
 interleave(seq6, seq7, result);  
 assert(result.size() == 3); // test that the properties of result are as they should be in the face of aliasing  
 assert(result.find("axe") == 0);  
 assert(result.find("borat") == 1);  
 assert(result.find("zebra") == 2);  
}