CS 32 Project 2

**Description**

I designed my Sequence class with multiple private member functions, as I wanted the public member functions to be relatively concise. I chose my private member functions to do several operations that would be used frequently by many public member functions. Each node contains a value, a pointer to the next node, and a pointer to the previous node. The first node in the list has a null pointer as its “next-node,” and the last node in the list has a null pointer as its “previous-node.” If there is only one node in the list, it contains a null pointer as both its “next-node” and “previous-node.” The head pointer points to the first node, and the tail pointer points to the last node. If the list is empty, both the head and tail pointers are null pointers. My doubly-linked list is not circular, and it does not contain any dummy nodes.

**Pseudocode**

This is my pseudocode for non-trivial algorithms:

Sequence(const Sequence &original) { //copy constructor

For each node in source,

Get value from source;

Copy value into target;

Set target’s size to source’s size;

}

~Sequence() { //destructor

If empty list,

Do nothing;

If 1 node in list,

Delete node;

If more than 1 node in list,

Delete each node;

}

Sequence& operator=(const Sequence &src) { //assignment operator

If target equals source,

Return target;

Clear memory from target;

Reset target’s size;

For each node in source,

Get value from source;

Copy value into target

Return target;

}

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

If invalid position,

Return -1;

Add node at position;

Increment size;

Return position;

}

int insert(const ItemType& value) {

If empty list,

Create new node at head;

Copy value;

Set next and prev to nullptr;

Set head and tail to new node;

If not empty list,

Find position;

Add node at position;

Increment size;

Return position;

}

bool erase(pos) {

If invalid position,

Return false;

Delete node;

Decrement size;

Return true;

}

int remove(const ItemType&value) {

For each node,

If value matches,

Delete node;

Increment count;

If value doesn’t match,

Increment position;

Return count;

}

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

If invalid position,

Return false;

Copy value;

Return true;

}

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

If invalid position,

Return false;

Update value;

Return true;

}

Int find(const ItemType& value) const {

For each node,

If value matches,

Return position;

Return -1;

}

void swap(Sequence& other) {

Swap sizes;

Swap heads;

Swap tails;

}

void setValue(int pos, const ItemType& value) { //private member function

Find position;

Update value;

}

void addNode(int pos, const ItemType& value) { //private member function

Create new node, copy value;

If empty list,

Set next and prev to nullptr;

Set head and tail to new node;

If first node in list,

Set next to next node, prev to nullptr;

Set head and second node’s prev to new node;

If last node in list,

Set prev to second-last node, next to nullptr;

Set tail and second-last node’s next to new node;

If between 2 nodes,

Find position;

Set next to next node, prev to previous node

Set previous node’s next and next node’s prev to new node;

}

void deleteNode(int pos) { //private member function

If empty list,

Return;

If only node in list,

Set head and tail to nullptr;

Delete node;

If first node in list,

Set head to second node;

Set second node’s prev to nullptr;

Delete node;

If last node in list,

Set tail to second-last node;

Set second-last node’s next to nullptr;

Delete node;

If between 2 nodes,

Find position;

Set previous node’s next to next node;

Set next node’s prev to previous node;

Delete node;

}

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

If invalid sizes,

Return -1;

Find position in seq1 where value equals first value of seq2;

If seq1 doesn’t contain first value of seq2,

Return -1;

If seq1 contains first value of seq2,

Until both lists are traversed,

If value of seq1 doesn’t equal value of seq2,

Find next position;

If value of seq1 equals value of seq2,

Get next values of seq1 and seq2;

If both lists are traversed,

Return position;

Break;

Return -1;

}

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

Create temp list;

If both lists are empty,

Set temp and result to empty list;

If 1 list is empty,

Set temp and result to other list;

Until both lists are traversed,

Until seq1 is traversed,

Get value of seq1;

Copy value into temp;

Increment positions;

Until seq2 is traversed

Get value of seq2;

Copy value into temp;

Increment positions;

Copy temp to result;

}

**Test Cases**

These are my test cases for my methods with ItemType set to std::string:

**int** main() {

Sequence a;

assert(a.empty() == **true**);

assert(a.empty()); //test empty()

assert(! a.empty() == **false**); //test empty()

cerr << "Test 1" << endl;

Sequence b;

assert(b.insert("1") == 0); //test insert()

Sequence c(b); //test copy constructor

ItemType elem = "";

assert(c.get(0, elem)); //test get()

assert(elem == "1");

assert(c.insert("2") == 1); //test insert()

assert(c.get(1, elem)); //test get()

assert(elem == "2");

assert(c.set(1, "1")); //test set()

assert(c.get(1, elem)); //test get()

assert(elem == "1");

assert(! c.get(2, elem)); //test get()

assert(c.size() == 2); //test size()

assert(c.erase(1)); //test erase()

assert(c.remove("1")); //test remove()

assert(c.size() == 0); //test size()

cerr << "Test 2" << endl;

Sequence d;

d = b; //test assignment operator

assert(d.get(0, elem)); //test get()

assert(elem == "1");

assert(d.insert("2")); //test insert()

assert(d.get(1, elem)); //test get()

assert(elem == "2");

assert(d.set(1, "1")); //test set()

assert(d.get(1, elem)); //test get()

assert(elem == "1");

cerr << "Test 3" << endl;

Sequence s;

s.insert("a"); //test insert()

s.insert("b"); //test insert()

s.insert("c"); //test insert()

s.insert("d"); //test insert()

s.insert("e"); //test insert()

s.insert("f"); //test insert()

s.insert("g"); //test insert()

Sequence t;

t.insert("d"); //test insert()

t.insert("e"); //test insert()

t.insert("f"); //test insert()

Sequence u;

u.insert("a"); //test insert()

u.insert("c"); //test insert()

cerr << "Test 4" << endl;

assert(subsequence(s, t) == 3); //test subsequence()

assert(subsequence(s, u) == -1); //test subsequence()

assert(subsequence(t, s) == -1); //test subsequence()

assert(subsequence(u, s) == -1); //test subsequence()

assert(subsequence(t, u) == -1); //test subsequence()

s.swap(t); //test swap()

assert(subsequence(t, s) == 3); //test subsequence()

s.swap(u); //test swap()

assert(subsequence(u, s) == -1); //test subsequence()

u.swap(s); //test swap()

assert(subsequence(t, s) == 3); //test subsequence()

cerr << "Test 5" << endl;

Sequence s1;

s1.insert(0, "paratha"); //test insert()

s1.insert(0, "focaccia"); //test insert()

Sequence s2;

s2.insert(0, "roti"); //test insert()

s1.swap(s2); //test swap()

//test size() and find()

assert(s1.size() == 1 && s1.find("roti") == 0 && s2.size() == 2 &&

s2.find("focaccia") == 0 && s2.find("paratha") == 1);

assert(s1.insert(1, "bread") == 1); //test insert()

assert(s1.get(1, elem)); //test get()

assert(elem == "bread");

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

assert(s1.insert(2, "paratha") == 2); //test insert()

assert(s1.get(2, elem)); //test get()

assert(elem == "paratha");

assert(s1.size() == 3); //test size()

assert(s1.insert(2, "focaccia") == 2); //test insert()

assert(s1.get(2, elem)); //test get()

assert(elem == "focaccia");

assert(s1.size() == 4); //test size()

cerr << "Test 6" << endl;

Sequence seq1;

ItemType value = "";

assert(seq1.empty()); //test empty()

seq1.insert(0, "b"); //test insert()

seq1.insert("d"); //test insert()

seq1.insert("c"); //test insert()

seq1.insert("a"); //test insert()

seq1.insert(3, "a"); //test insert()

seq1.set(1, "e"); //test set()

assert(seq1.size() == 5); //test size()

seq1.remove("a"); //test remove()

assert(seq1.find("d") == 2); //test find()

assert(seq1.size() == 3); //test size()

seq1.get(0, value); //test get()

assert(value == "e");

seq1.get(1, value); //test get()

assert(value == "c");

seq1.get(2, value); //test get()

assert(value == "d");

seq1.get(3, value); //test get()

assert(value == "d");

seq1.get(4, value); //test get()

assert(value == "d");

seq1.remove("b"); //test remove()

assert(seq1.size() == 3); //test size()

seq1.erase(1); //test erase()

assert(seq1.size() == 2); //test size()

seq1.remove("d"); //test remove()

assert(seq1.size() == 1); //test size()

seq1.erase(0); //test erase()

assert(seq1.size() == 0); //test size()

cerr << "Test 7" << endl;

Sequence f;

f.insert("A"); //test insert()

f.insert("B"); //test insert()

f.insert("C"); //test insert()

Sequence g;

g.insert("aa"); //test insert()

g.insert("bb"); //test insert()

g.insert("cc"); //test insert()

g.insert("dd"); //test insert()

g.insert("ee"); //test insert()

g.insert("ff"); //test insert()

Sequence h;

cerr << "Test 8" << endl;

interleave(f, g, h); //test interleave()

ItemType term;

h.get(0, term); //test get()

assert(term == "A");

h.get(1, term); //test get()

assert(term == "aa");

h.get(2, term); //test get()

assert(term == "B");

h.get(3, term); //test get()

assert(term == "bb");

h.get(4, term); //test get()

assert(term == "C");

h.get(5, term); //test get()

assert(term == "cc");

h.get(6, term); //test get()

assert(term == "dd");

h.get(7, term); //test get()

assert(term == "ee");

h.get(8, term); //test get()

assert(term == "ff");

cerr << "All tests passed" << endl;

**return** 0;

}