Edward Chu 504772158

CS 32 Project 2 - Report

1. Doubly-Linked List Implementation

Each sequence contains a pointer to the first node and the last node of the sequence, and also a int storing the size of the sequence (number of nodes). An empty sequence’s pointers will both be nullptrs, and the size will be 0. A sequence with only one node will have both pointers pointing to the same node, with the size being 1. The list itself is not circular, but linear.

Node is included as a nested class to sequence, which contains the value of the node (ItemType to be stored), a pointer to the previous node and a pointer to the next node. The head node will have a nullptr previous pointer, while the tail node will have a nullptr next node.

2. Pseudocode for Non-Trivial Functions

Sequence::~Sequence(){

While the current node is not a nullptr

Delete the current node

}

Sequence::Sequence(Sequence const &other){

Initialize as empty sequence

For each node in other sequence

Insert it into this sequence

}

Sequence& Sequence::operator= (const Sequence &other){

If both sequences are not the same

Empty out this sequence

For each node in other seq

Insert it into this seq

Return pointer to this sequence

}

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

If pos is out of bounds

Return false

If list is empty

Create a new node

Make the new node head node and tail node of the sequence

Add 1 to seq size

Return true

If insert into front

Create a new node with nullptr as prevNode and old head node as nextNode

Make the old head node’s prevNode point to the new node

Make the new node the new head node

Add 1 to seq size

Return true

If insert into back

Create a new node with old tail node as prevNode and nullptr as nextNode

Make the old tail node’s nextNode point to the new node

Make the new node the new tail node

Add 1 to seq size

Return true

Else when inserting into middle

Find suitable position to insert new node

Create a new node with the original node’s prevNode as the prevNode and the original node as the nextNode

The original’s prevNode’s nextNode now points to the new node

The original’s prevNode now points to the new node

Add 1 to seq size

Return true

}

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

For each node

If insertValue <= currentNode’s value

Insert using insert(pos, value)

Return pos

Otherwise, insert value at the end

Return newSeqSize - 1

}

bool Sequence::erase(int pos){

If Out of Bounds pos

Return false

If only 1 node in seq

Delete the only node

Set headNode and tailNode pointers to be nullptr

Decrement size

Return true

If erasing first node

Delete head node

Set old Second Node’s prevNode to be nullptr

Set the new headNode to be the old second node

Decrement size

Return true

If erasing last node

Delete last node

Set old Second-to-last Node’s nextNode to be nullptr

Set the new tailNode to be the old second-to-last node

Decrement size

Return true

Otherwise

Search for the correct pos

Join the toBeRemoved node’s next node and previous node together

Delete the toBeRemoved node

Decrement size

Return true

}

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

Intialize removeCount to 0

For each node

If value of node == value to be erased

Call erase(pos) on this node

Increment count

Return removeCount

}

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

If out of bounds pos

Return false

Otherwise, search for the correct pos

Set value to the current pos’s value

Return true

Return false, but this line should never be reached

}

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

If out of bounds pos

Return false

Otherwise, search for the correct pos

Set the current node’s value to value

Return true

Return false, but this line should never be reached

}

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

For each node

If current node’s value == value

Return this pos

Return -1 when no node has this value

}

void Sequence::swap(Sequence& other){

Swap pointers to head nodes

Swap pointers to tail nodes

Swap seq sizes

}

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

If either seq size == 0

Return false

For each node in seq1

Compare against first node in seq2, if equal

For each sequential node in seq1 and seq2, compare if they are equal, break out if false

If end of seq2 is reached, return start of subseq

Return -1, subseq not found

}

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

{

Create empty sequence save

While end of seq1 and seq2 are not reached

Insert Node from seq1 to save

Insert Node from seq2 to save

result = save

}

3. Tests

These were performed on a sequence of strings (i.e. ItemType def as std::string)

Sequence a;

assert(a.empty());

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

//Testing insert(pos, item)

a.insert(0, "Apples"); //Empty

a.insert(0, "Aaa"); //Insert start

a.insert(2, "Cheerios"); //Insert end

a.insert(2, "Bananas"); //Insert middle

a.insert(4, "Doritos"); //Insert end

a.dump();

//Testing insert(item)

assert(a.insert("Abc") == 1); //Insert middle

assert(a.insert("Aa") == 0); //Insert start

assert(a.insert("Elephants") == 7); //Insert end

a.dump();

//Testing erase

assert(!a.erase(-1)); //Out of bounds

assert(!a.erase(8)); //Out of bounds

assert(a.erase(0)); //Delete first item

assert(a.erase(1)); //Delete middle item

assert(a.erase(5)); //Delete end item

a.dump();

//Testing remove

a.insert("Aaa");

assert(a.remove("Zebras") == 0); //Remove non-existent item

assert(a.remove("Aaa") == 2); //Remove item, contiguous

a.insert(4, "Apples");

assert(a.remove("Apples") == 2); //Remove item, non-contiguous

a.insert(0, "Apples");

a.dump();

a.insert(0, "Remove");

a.insert(2, "Remove");

a.insert(6, "Remove");

assert(a.remove("Remove") == 3); //Remove multiple items, non-contiguous

a.dump();

//Testing get

string test;

assert(a.get(0, test) && test == "Apples"); //Check normal

assert(a.get(1, test) && test == "Bananas"); //Check normal

assert(a.get(2, test) && test == "Cheerios"); //Check normal

assert(a.get(3, test) && test == "Doritos"); //Check normal

assert(!a.get(-1, test) && test == "Doritos"); //Out of bounds

assert(!a.get(4, test) && test == "Doritos"); //Out of bounds

//Tesing set

assert(a.set(0, "Anas")); //Check normal

assert(a.get(0, test) && test == "Anas");

assert(a.set(1, "Banas")); //Check normal

assert(a.get(1, test) && test == "Banas");

assert(!a.set(-1, test)); //Out of bounds

assert(!a.set(4, test)); //Out of bounds

//Testing find

a.dump();

assert(a.find("Anas") == 0); //Non-existent item

assert(a.find("Banas") == 1); //Check normal

assert(a.find("Doritos") == 3); //Check normal

a.insert(4, "Banas");

assert(a.find("Banas") == 1); //Return first instance of Banas

a.dump();

//Testing Swap

Sequence b;

b.insert("Hi");

a.swap(b); //Check Normal

b.swap(b); //Swap with self

a.dump();

b.dump();

//Testing Copy Constructor

Sequence c(b); //Check normal

c.dump();

Sequence d;

Sequence e(d); //Copy empty sequence

e.dump();

//Testing Assignment Operator

a = c; //Check normal

a = a; //Assign to self

a.dump();

b.set(0, "Banas");

c.set(0, "Canas");

assert(a.get(0, test) && test == "Anas"); //Check that after assignment operator, sequences are pointing to different lists

assert(b.get(0, test) && test == "Banas"); //Ditto for copy constructor

assert(c.get(0, test) && test == "Canas"); //Ditto for assignment operator

//Testing new Functions

Sequence uno;

Sequence dos;

uno.insert("5");

uno.insert("6");

uno.insert("7");

uno.insert("8");

uno.insert("9");

uno.dump();

dos.insert("7");

dos.insert("8");

dos.insert("9");

dos.dump();

//Testing subsequence

assert(subsequence(uno, dos) == 2); //Check normal, subsequence at end

dos.set(2, "8");

assert(subsequence(uno, dos) == -1); //Check non-existent subsequence

assert(subsequence(dos, uno) == -1); //Error when seq2 is longer than seq1

dos.set(0, "6");

dos.set(1, "7");

dos.set(2, "8");

assert(subsequence(uno, dos) == 1); //Check normal, subsequence in middle

dos.set(0, "5");

dos.set(1, "6");

dos.set(2, "7");

assert(subsequence(uno, dos) == 0); //Check normal, subsequence at the start

dos = uno;

assert(subsequence(uno, dos) == 0); //Check identical sequence are subsequence

Sequence testSub1;

Sequence testSub2;

testSub1.insert(0, "77");

testSub1.insert(1, "77");

testSub1.insert(2, "77");

testSub1.insert(3, "78");

testSub2.insert(0, "77");

testSub2.insert(1, "78");

assert(subsequence(testSub1, testSub2) == 2); //Check when seq1 have multiples of items in seq2

Sequence test1;

Sequence test2;

test1.insert(0, "30");

test1.insert(1, "21");

test1.insert(2, "63");

test1.insert(3, "42");

test1.insert(4, "17");

test1.insert(5, "63");

test2.insert(0, "42");

test2.insert(1, "63");

test2.insert(2, "84");

test2.insert(3, "19");

Sequence answer;

interleave(test1, test2, answer); //Check normal

assert(answer.get(0, test) && test == "30");

assert(answer.get(2, test) && test == "21");

assert(answer.get(8, test) && test == "17");

assert(answer.get(9, test) && test == "63");

answer.dump();

Sequence empty;

interleave(answer, empty, answer); //Check when interweaving with empty sequence

assert(answer.get(0, test) && test == "30");

assert(answer.get(2, test) && test == "21");

assert(answer.get(8, test) && test == "17");

assert(answer.get(9, test) && test == "63");

interleave(answer, answer, answer); //Check when interweaving with self

assert(answer.get(0, test) && test == "30");

assert(answer.get(4, test) && test == "21");

assert(answer.get(16, test) && test == "17");

assert(answer.get(19, test) && test == "63");

answer.dump();

cerr << endl << "All tests succeeded" << endl;

The following tests were performed using unsigned long as ItmeType

Sequence a;

a.insert(0, 0); //Testing normal

a.insert(1, 1);

a.insert(2, 2);

a.insert(3, 3);

ItemType test;

assert(a.get(2, test)); //Testing get

assert(!a.get(4, test)); //Out of bounds

assert(test == 2);

assert(a.erase(2)); //Test erase

//a.dump();

assert(a.get(2, test)); //Make sure erase works

assert(test == 3);

Sequence b;

b.insert(5);

b.insert(6);

b.insert(7);

b.insert(8);

//b.dump();

a.swap(b); //Test swap

a.dump();

b.dump();

assert(a.get(1, test));

assert(!b.get(3, test));

assert(test == 6);

//b.insert(1.5);

//assert(b.remove(1.5) == 2);

//b.dump();

b.insert(2, 1);

unsigned long temp;

assert(b.get(2, temp));

assert(temp == 1);

assert(b.find(1) == 1); //Test find

assert(b.find(10) == -1); //Finding non-existent item

assert(b.erase(1)); //Test erase

assert(b.get(1, test) && b.get(2, temp));

assert(test == 1 && temp == 3);

b.set(2, 1); //Testing set

assert(b.get(2, test));

assert(test == 1);

b.insert(4);

assert(b.find(4) == 3); //Testing find, multiple instance of same item

b.insert(0); //Testing insert at start

assert(b.get(0, test) && b.get(1, temp));

assert(test == 0 && temp == 0);

for(int i = 3; i < 6; i++)

b.erase(3);

a.swap(b); //Testing swap

assert(!a.get(4, temp));

assert(a.get(2, temp));

assert(temp == 1);

//Bad

assert(!a.insert(10, 2)); //Out of bounds

assert(!a.insert(-1, 2)); //Out of bounds

Sequence c(a); //Copy constructor

a.dump();

c.dump();