Joel George

Project 2 Report

**Linked List Design Explanation:**

My doubly linked list is linear, with each node featuring a next pointer, a previous pointer, and a value of ItemType. The list also has a head pointer and a tail pointer.

Regular Sequence (linked list diagram):

address: 990

value: *X*

next: *1000*

prev: *nullptr*

address: 1010

value: *Z*

next: *nullptr*

prev: *1000*

address: 1000

value: *Y*

next: *1010*

prev: *990*

*head*

*tail*

**Empty Sequence:**

*head*

*tail*

*nullptr*

*nullptr*

**Pseudocode:**

uncheckedInsert (the private function I used to handle insertions into the list):

*create new node and assign value into it*

*if the list is empty{*

*add the node*

*set its previous and next pointers to nullptr*

*increment size of list}*

*else if inserting at end of list{*

*set the tail to point to the new node*

*update the new node’s previous and next pointers}*

*else if inserting at the beginning of a list{*

*set the head to point to the new node*

*update the new node’s previous and next pointers}*

*else if inserting node in the middle of the list{*

*traverse through list until pointing at desired position*

*add node and update its previous and next pointers}*

erase:

*if position is out of bounds{*

*return false}*

*if erasing a list’s only node{*

*delete the node*

*set head and tail to nullptr, decrement size*

*return true}*

*else if erasing node at the end of a list{*

*delete the node*

*move the tail to the previous node*

*decrement size*

*return true}*

*else if erasing node in middle{*

*traverse through list until pointing at node in desired position*

*link the node that precedes the current node and the node that follows it*

*delete the node*

*decrement size*

*return true}*

**Test Cases:**

remove:

*traverse through list*

*if value of node equals parameter value{*

*call erase on that position*

*increment numRemoved*

*decrement i}*

*return numRemoved*

swap:

*Copy current head into pointer to Node*

*Copy other sequence’s head into current head*

*Copy pointer to Node into other sequence’s head*

*Repeat procedure for tail and list size*

Copy constructor:

*Traverse through src list*

*If at position 0{*

*Create new Node and make the head and tail point to it*

*Set value to value of corresponding node in source sequence*

*Set prev and next pointers to nullptr*

*Increment pointer traversing source list, and set pointer traversing through current list to head}*

*Else{*

*Create new Node and update the node before it (update its next pointer)*

*Set value to value of corresponding node in source sequence*

*Set prev and next pointers to nullptr*

*Increment pointer traversing source list, and set pointer traversing through current list to current node}*

Assignment Operator:

*If src sequence is equal to the current sequence*

*Do nothing*

*Else*

*Delete all nodes in the existing sequence*

*Set head and tail to nullptr*

*Essentially, do same thing as copy constructor (traverse through src sequence and copy nodes into current sequence)*

Subsequence:

*If size of seq2 is greater than seq1 or if seq2 is empty*

*Return -1*

*While position does not go out of bounds for either seq1 or seq2{*

*Copy the value of seq1 at position j*

*Copy the value of seq2 at position i*

*If the two values are equal, increment i, j, and count variable*

*Else*

*If at the beginning of seq2*

*Increment j*

*Go to the beginning of seq2*

*Set count to 0*

*}*

*If count does not equal the size of seq2*

*Return -1;*

*Else*

*Return the position that the subsequence starts at*

Interleave

*Set count to 1, j to 0, and k to 0*

*While int i is less than the sum of the sizes of the two sequences*

*If count equals 1*

*If j is greater than seq1 size{*

*Copy node from seq2 into temporary sequence*

*Increment k }*

*Else {*

*Copy node from seq1 into temporary sequence*

*Increment j*

*Increment count to 2}*

*If count equals 2*

*If j is greater than seq2 size{*

*Copy node from seq1 into temporary sequence*

*Increment j}*

*Else {*

*Copy node from seq2 into temporary sequence*

*Increment k}*

*Decrement count to 1*

*Set result equal to temporary sequence*

**Test Cases (for unsigned long):**

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

To test that insert fails if passed a position out of bounds (i.e. negative or greater than the size of the sequence):

Sequence a;

assert(a.insert(1, 42) == false);

assert(a.insert(-1, 42) == false);

To test that insert works correctly on an empty list:

assert(a.insert(0, 24) == true);

To test that insert correctly inserts value at end of string:

assert(a.insert(1, 43) == true);

To test that insert correctly inserts value at beginning of string:

assert(a.insert(0, 49) == true);

To test that insert correctly inserts value in middle of string:

assert(a.insert(1, 55) == true);

*insert(const ItemType& value)*

To make sure insert works with empty list:

Sequence b;

assert(b.insert(4) == 0);

To make sure insert works with already-filled list:

assert(b.insert(7) == 1);

To make sure insert correctly sorts values:

assert(b.insert(5) == 1);

assert(b.insert(7) == 2);

assert(b.insert(10) == 4);

erase(int pos):

To make sure erase returns false when given out of bounds position:

Sequence c;

assert(erase(1) == false);

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

To make sure erase works correctly on a list of size 1:

c.insert(5);

assert(erase(0) == true);

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

To make sure erase correctly works on first node of list:

c.insert(5);

c.insert(5);

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

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

To make sure erase works correctly on last node of list:

c.insert(5);

assert(erase(1) == true);

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

To make sure erase works correctly on node in the middle:

c.insert(4);

c.insert(6);

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

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

remove(const ItemType& value):

To make sure that remove doesn’t remove any nodes when the passed in value is not present in the sequence:

Sequence d;

d.insert(1);

d.insert(2);

d.insert(2);

assert(d.remove(5) == 0);

To make sure that remove removes all instances of a value:

assert(d.remove(2) == 2);

assert(d.size() == 1 && d.find(1) == 0);

get(int pos, ItemType& value) const:

To test that get returns false if passed in an out of bounds position:

Sequence e;

e.insert(1);

ItemType value;

assert(e.get(1, value) == false);

To test that get sets value to the right value:

assert(e.get(0, value) == true);

assert(value == 1);

set(int pos, const ItemType& value):

To test that set returns false if passed in an out of bounds position:

assert(e.set(1, 2) == false);

To test that set correct sets node value to passed in value:

assert(e.set(0, 2) == true);

assert(e.get(0, value) == true);

assert(value == 2);

find(const ItemType& value) const:

To make sure find returns -1 if the value is not present in the sequence:

Sequence f;

f.insert(2);

assert(f.find(3) == -1);

To make sure find returns position of value if found:

assert(f.find(2) == 0);

To make sure find returns position of *first* instance of value:

f.insert(2);

assert(f.find(2) == 0 && f.size() == 2);

swap(Sequence& other):

To make sure that swap works with empty nodes:

Sequence g;

Sequence h;

g.swap(h);

assert(g.size() == 0 && h.size() == 0);

To make sure that swap works with one empty list and one full list:

g.insert(1);

g.swap(h);

assert(g.size() == 0 && h.size() == 1 && h.find(1) == 0);

To make sure that swap works on two full lists:

g.insert(2);

g.swap(h);

assert(g.size() == 1 && h.size() == 1 && g.find(1) == 0 && h.find(2) == 0);

To make sure that swap works on lists of different sizes:

g.insert(3);

g.swap(h);

assert(g.size() == 1 && h.size() == 2 && h.find(3) == 1 && g.find(2) == 0);

To make sure that function works in the face of aliasing:

g.swap(g);

assert(g.size() == 1 && g.find(2) == 0);

Sequence(const Sequence &src):

To make sure that function works with empty Sequence:

Sequence i;

Sequence j = i;

To make sure that function works with full Sequence:

j.insert(4);

Sequence k = j;

assert(k.size() == 1 && k.find(1) == 0);

operator=(const Sequence &src):

To make sure that operator works in the face of aliasing:

Sequence l;

l = l;

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

To make sure that operator works when first sequence has preexisting nodes:

l.insert(4);

l.insert(5);

l.insert(6);

Sequence k;

k.insert(1);

k.insert(2);

l = k;

assert(l.size() == 2 && l.find(1) == 0 && l.find(2) == 1);

To make sure that the operator works when the second sequence is empty:

Sequence m;

l = m;

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

To make sure that the operator works when the original sequence is empty:

l = k;

assert(l.size() == 2 && l.find(1) == 0 && l.find(2) == 1);

subsequence(const Sequence& seq1, const Sequence& seq2):

To make sure that function correctly returns -1 when seq2 is empty:

Sequence n;

Sequence o;

o.insert(1);

o.insert(2);

o.insert(3);

assert(subsequence(o, n) == -1);

To make sure that function correctly returns -1 if seq2 is bigger than seq1:

n.insert(1);

n.insert(2);

assert(subsequence(n, o) == -1);

To make sure that function correctly returns position if seq2 is subsequence of seq1:

Sequence p;

p.insert(1);

p.insert(2);

assert(subsequence(o, p) == 0);

To make sure that function correctly returns *first* position that subsequence is found:

o.insert(1);

o.insert(2);

assert(subsequence(o, n) == 0);

Sequence q;

q.insert(3);

q.insert(1);

q.insert(2);

assert(subsequence(o, q) == 2);

To make sure that function correctly works in the face of aliasing:

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

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

To make sure that function correctly “interleaves” empty sequences:

Sequence r;

Sequence s;

Sequence t;

interleave(s, t, r);

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

To make sure function correctly interleaves full sequences:

s.insert(1);

s.insert(2);

t.insert(3);

t.insert(4);

interleave(s, t, r);

std::cout << “List contents of R” ;

r.dump(); //confirm visually (should be 1 3 2 4)

To make sure interleave returns a copy of the full sequence if one of the sequences is empty:

Sequence u;

interleave(u, s, r);

std::cout << “List contents of R” :;

r.dump(); //confirm visually (should be 1 2)

To make sure interleave works, even if result has its own values:

Sequence v;

v.insert(42);

v.insert(43);

v.insert(44);

interleave(s, t, v);

std::cout << “List contents of V” ;

v.dump(); //confirm visually (should be 1 3 2 4)

To make sure interleave works in the face of aliasing:

interleave(s, t, s);

std::cout << “List contents of S” ;

s.dump(); //confirm visually (should be 1 3 2 4)

interleave(s, s, s);

std::cout << “List contents of S” ;

s.dump(); //confirm visually (should be 1 1 3 3 2 2 4 4)