My Doubly-Linked List

I used a fairly simple, yet effective doubly-linked list in my project. To start, I initialized the head and tail pointers to nullptr and used conditional statements to mark the head and tails of the Sequence. Of course, each node had next and previous pointers, where the head pointer’s previous was nullptr and the tail pointer’s next was nullptr. As you can see, this was not a circular linked list. The erase function also accounts for the head and tail pointers with conditional statements. The destructor simply deleted every node and set the head and tail to nullptr.

*Picture of possible Sequence below:*

nullptr

nullptr

next

TAIL

NODE

HEAD

next

next

previous

previous

previous

Pseudocode

Sequence(const Sequence& other)

{

head and tail = nullptr;

initialize pos variable to 0

size = other size

initialize node variable (n) to other head

traverse other seqeunce using n

insert value of n into pos

increment pos

}

~Sequence()

{

initialize node variable (n) to other head

traverse entire seqeunce using n

delete n

head and tail = nullptr

}

operator=(const Sequence& other)

{

Destruct sequence

Construct new sequence using other (copy constructor)

return newly constructed Sequence

}

insert(int pos, const ItemType& value)

{

if pos is out of range

return false;

if pos is beginning of linked list

{

create new node

store value in node

next pointer connects to head

previous pointer is nullptr

node becomes head

if size is not zero

node after’s previous pointer points to this node

if first node is last node

node becomes tail

increment size

return true

}

if pos is end of linked list

{

traverse list to last node (p)

create new node (n)

store value

p's next points to new node

n's'previous points to last node

n's next is nullptr

n becomes tail

increment size

return true

}

if pos is in middle of list

{

traverse list to node before position of insertion (p)

create new node (n)

store value in n

n's next points to node after p

p's next becomes n

node after n's previous becomes n

n's previous becomes p

increment size

return true

}

return true

}

insert(const ItemType& value)

{

create node variable initialized at head

initialize count variable to 0

while traversing through entire linked list

{

if value of node at count is greater than value

insert new node containing value at count

return true

increment count

}

if count reaches size

insert new node containing value at end of linked list

return count

}

erase(int pos)

{

if pos is out of bounds

return false;

if pos is at only node

{

delete head pointer

head and tail = nullptr

decrement size

return true

}

if pos is at beginning

{

initialize node variable to head (n)

head becomes node after n

delete n

head's previous pointer = nullptr

decrement size

return true

}

if pos is in middle of list

{

initialize node variable to head (n)

traverse to pos using n

node before n's next becomes n's next

node after n's previous becomes n's previous

delete n

decrement size

return true

}

if position is last element

{

initialize node variable to tail(n)

node before n becomes tail

tail's next pointer = nullptr

delete n

decrement size

return true

}

return true

}

remove(const ItemType& value)

{

initialize numRemoved variable to 0

while value is found in list

erase position where value is found

increment numRemoved

return numRemoved

}

get(int pos, ItemType& value) const

{

if pos is out of range

return false

initialize node variable to head (n)

traverse linked list to pos

store n's value into value

return true

}

set(int pos, const ItemType& value)

{

if pos is out of range

return false

initialize node variable to head (n)

traverse linked list to pos

store value into n's value

return true

}

find(const ItemType& value) const

{

initialize node variable to head (n)

initialize count variable to 0

traverse linked list using n

if value at n = value

return count

increment count

return -1 if value not found

}

swap(Sequence& other)

{

Copy contruct temporary sequence variable using this pointer

Using assignment operator, assign sequence to other

Assign the temporary sequence to other

}

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

{

fi seq2 is empty or greater in size than seq1

return -1

initialize boolean variable to false (sequence)

declare ItemTypes for each sequence and an int startingPos

traverse through first sequence (variable k)

if value of seq1 at k is equal to first value of seq2

startingPos = k

new variable for starting position initialized at k (newPos)

traverse through seq2 (variable w)

sequence = true

if values do not match

sequence = false

break

increment newPos and w

if sequence = true

return startingPos

return -1 (no subsequence found)

}

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

{

construct temporary sequence from result (temp)

erase all elements from temp

if seq1 is empty

temp = seq2

result = temp;

return

if seq2 is empty

temp = seq1

result = temp;

return

create size variables initialized to seq1 and seq2 sizes (oneCount, twoCount, tempCount)

create count variables for seq1, seq2, and temp initialized to 0

while the larger sequence is not fully traversed

{

If sequence one is not fully traversed

Store value at position oneCount

Insert value at tempCount and increment both counts

If sequence two is not fully traversed

Store value at position twoCount

Insert value at tempCount and increment both counts

}

result = temp

return

}

Test Cases

Sequence l; *default constructor*

assert(l.insert(0, "lavash")); *test Insert*

assert(l.insert(0, "tortilla")); *test Insert*

assert(l.size() == 2); *test Size*

ItemType x = "injera";

assert(l.get(0, x) && x == "tortilla"); *test Get*

assert(l.get(1, x) && x == "lavash"); *test Get*

Sequence s;

assert(s.empty()); *test empty*

assert(s.find("laobing") == -1); *test empty/can’t find*

s.insert("laobing");

assert(s.size() == 1 && s.find("laobing") == 0); *test insert and find*

s.insert("aaa"); *test insert/insertion place*

assert(s.size() == 2 && s.find("aaa") == 0); *test location of insert*

s.insert("laobingboass");

assert(s.size() == 3 && s.find("laobingboass") == 2);

s.insert(1, "yard");

assert(s.size() == 4 && s.find("yard") == 1 && s.find("aaa") == 0); *test insert*

assert(s.find("laobing") == 2 && s.find("laobingboass") == 3); *test find and insert*

Sequence a;

assert(a.empty() && a.size() == 0);  *test empty and size*

assert(!a.insert(2, "yaes")); *pos out of range*

assert(a.insert(0, "rajiv") && a.find("rajiv") == 0); *pos in range*

assert(!a.erase(1)); *pos out of range/test erase*

assert(!a.set(-1, "yard")); *pos out of range/test set*

assert(a.set(0, "lard") && a.find("lard") == 0);

assert(a.erase(0) && a.find("lard") == -1);  *test erase*

Sequence num;

num.insert(0, 50);

num.insert(0, 50);

num.insert(0, 50);

num.insert(0, 50);

num.insert(0, 27);

num.insert(0, 28);

num.insert(0, 50);

assert(num.erase(0));

assert(num.erase(2));

assert(num.erase(4));

assert(num.remove(50) == 2);  *Test erase and remove*

Sequence b;

assert(b.insert(0, "chump"));

assert(b.insert("boo") == 0);

s.swap(b);

assert(s.size() == 2 && s.find("boo") == 0 && s.find("chump") == 1);

assert(b.size() == 4 && b.find("aaa") == 0 && b.find("laobingboass") == 3 &&

b.find("laobing") == 2); *Test swap function*.

Sequence empty;

b.swap(empty);

assert(b.size() == 0 && empty.size() == 4); *Test swap with empty sequence*

Sequence comp;

assert(comp.insert(0, 31));

assert(comp.insert(1, 21));

assert(comp.insert(2, 63));

assert(comp.insert(3, 42));

assert(comp.insert(4, 17));

assert(comp.insert(5, 63));

assert(comp.insert(6, 17));

assert(comp.insert(7, 29));

assert(comp.insert(8, 8));

assert(comp.insert(9, 32));

Sequence comp2; *default constructor*

assert(comp2.insert(0, 63));

assert(comp2.insert(1, 17));

assert(comp2.insert(2, 29));

assert(subsequence(comp, comp2) == 5); *Test subsequence function*

assert(subsequence(comp2, comp) == -1); *Test subsequence for failure*

Sequence result = comp;

interleave(comp, comp2, result);

assert(result.size() == 13); *Test interleave*

interleave(comp2, comp2, comp2);

assert(comp2.size() == 6); *Test interleave w/ aliasing*

Sequence emptyTwo;

interleave(emptyTwo, comp2, emptyTwo);

assert(emptyTwo.size() == 6); *Test interleave w/ empty sequence*