005994409

Timothy Liu

The doubly linked list I’m using holds an ItemType, a pointer to the next node, and a pointer to the previous node. A circular doubly linked list was not used. So, when there are 0 elements the head and tail pointers will be pointing to nullptrs. When there is a singular element, the head and tail pointers will be pointing to the same node. When there are multiple elements, each node will store a pointer to the previous node and a pointer to the next node, with the head node’s pointer to the previous node being a nullptr and the tail node’s pointer to the next node being a nullptr. The tail node’s next pointer will also point to a nullptr. The head is a pointer to the first node representing the first element in the sequence and the tail is a pointer to the last node representing the last element in the sequence.

Diagram

Description automatically generated

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

Check if position in bounds

If pos out of bounds, return -1

If linked list is empty, create the first node

Create new node with value

Set head and tail to the new node

Set next and prev of new node to null

Increment sequence size

Return pos

Otherwise create new node for value

Iterate through linked list until you’re at one node before desired position

Initialize node pointer to head

Move p to next node if you’re not there yet

If p points to head

Insert new node at beginning

Update head pointer

If p points to null

Insert new node at end

Update tail pointer

Otherwise

Insert new node between existing nodes

Increment sequence size

Return position

**int insert(ItemType& value)**

insertedPosition set to 0 to keep track of where the value gets inserted

Create new node with value

If linked list is empty

Set head and tail to the new node

Set next and prev of new node to nullptr

Increment sequence size

Return insertedPosition

Otherwise start at first node by initializing p to head

Iterate through nodes until you reach end or you find the position where the new node value is smaller than the current node value

Move p to the next node

Increment insertedPosition to reflect current index

If node should be inserted to beginning where head currently is

Insert new node at beginning

Update head pointer

If node should be inserted to very end where tail currently is

Insert new node at end

Update tail pointer

Otherwise

Insert new node between existing nodes

Increment sequence size

Return insertedPosition

**bool erase(int pos)**

Check if pos within bounds or sequence is empty

Return false

If there’s only one node in the list

Decrement sequence size

Delete head

Set head and tail to null

Return true

Otherwise go through linked list until you find node to be removed at the desired index position starting at head

If head needs to be removed

Set new head’s prev to null

Update head pointer

If tail needs to be removed

Set new tail’s next to nullptr

Update tail pointer

Otherwise node in-between needs to be removed

Update previous node’s next

Update next node’s prev

Delete node you’re currently pointing at

Decrement sequence size

Return true

**remove( Itemtype& value )**

Initialize removedItems to 0

If list is empty, return 0

Iterate through nodes starting at head and check if node’s value is equal to value you want to remove

If the node is head then update next node to be the new head

If the node is tail then update the previous node to be the new tail

Otherwise update the previous node’s next and next node’s previous so they’re connected

Delete the current node, decrement the size, and increase the # of removed items

Return the # of nodes removed

**bool get(int pos, ItemType& value)**

Check if pos out of bounds

Return false

Starting at head, iterate through nodes until you reach the node you want

Copy the node’s stored value into the value you want

Return true

**bool set( int pos, ItemType& value )**

If pos out of bounds or size is 0, return false

Iterate through the nodes starting at head until you reach the node you want

At the current node, change the node’s stored value to the new value

Return true

**int find( ItemType& value )**

Indicate that value hasn’t been found yet

Iterate through all the nodes starting at head

If a node’s value is the same as the value you want to find

Indicate that value has been found

Record the position of that node

Otherwise, go to next node

If value has been found, return the position of the first node that contains the value

Otherwise if value hasn’t been found return -1

**void swap( Sequence& Other )**

Store other’s size in a temp

Copy this size into other’s size

Copy other’s size (held in a temp) into this size

Store other’s head into a temp

Have other’s head point to the same node as this head

Have this head point to the same node as other’s head was pointing originally

Store other’s tail into a temp

Have other’s tail point to the same node as this tail

Have this tail point to the same node as other’s tail was pointing originally

**Assignment Operator**

If not equal

Create temporary sequence with exact characteristics of righthand side sequence (w/ swap)

Put temporary sequence into the one you want

Return current sequence

**Copy Constructor**

If empty sequence

Set head and tail to nullptr

Return early

Create new head node

Copy over other head’s node value into the new sequence’s head node value

Have new sequence’s head’s next and prev point to nullptr

Set tail to head

Initialize pointer to other head’s next node to go through other sequence

Initialize another pointer to new sequence’s head to go through new sequence

Go through the nodes of the other sequence until you reach end

Dynamically create a new Node

Copy over other node’s value into new node

Set this new node’s next to nullptr and prev to the node before

Have new sequence’s pointer next point be to the new Node

Set tail to the newly created node

Continue onto other sequence’s next node

**void concatReverse(const Sequence& seq1, const Sequence& seq2, Sequence& result)**

Clear the result sequence to size 0

Insert seq1 into result while also reversing

Iterate through seq1 and getting the value each time starting from the front

Append the value each time to the start of the result sequence

Insert seq2 into result after seq1 while also reversing

Iterate through seq2 and getting the value each time starting from the front

Append the value each time to the result sequence after where seq1 was appended

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

Bounds check

Iterate through seq1

Set index for going through seq2

Iterate through seq2

Set subsequence start position preemptively

Get temp variable to compare seq1 and seq2 values

If no match at current corresponding indices, break iteration

If able to iterate through seq2 entirely, then subsequence found and return early

Otherwise, seq1 has been iterated through with no subsequence was found

Test Cases:

Sequence empty;

assert( empty.size() == 0 && empty.empty() ); // Empty sequence

assert( empty.erase(0) == **false** ); // Erasing from 0 size sequence

assert( empty.remove("nothing") == 0 ); // Removing from 0 size sequence

assert( empty.set(0, "Still nothing") == **false** ); // Setting value of empty sequence

assert( empty.set(1, "Still nothing") == **false** ); // Setting value out of bounds

Sequence noLongerEmpty(empty); // Copy constructor on empty sequence

assert( noLongerEmpty.size() == 0); // Check if size same

Sequence a;

assert( a.insert("c") == 0); // Insert without position

assert( a.insert("a") == 0); // Alphanumerical insertion

assert( a.insert("b") == 1); // Alphanumerical insertion

assert( a.size() == 3 ); // Size after insertion

assert( a.empty() == **false** ); // Empty or not after insertion

Sequence b;

b = a; // Assignment operator

assert( b.size() == 3 ); // Check size after assignment

assert( b.find("c") == 2 ); // Check whether sequence still the same after assignment

assert( b.find("b") == 1 );

assert( b.find("a") == 0 );

assert( b.set(0, "1") == **true** ); // Setting 0th position

assert( b.size() == 3 ); // Check size

assert( b.insert(0, "2") == 0 ); // Insert new value into new sequence object

assert( a.find("a") == 0 ); // Check if original sequence isn't altered

assert( b.erase(0) == **true**); // Erase from sequence size larger than 1

assert( b.erase(0) == **true**);

assert( b.erase(1) == **true**); // Erase last value from sequence

assert( b.erase(0) == **true**); // Erase from sequence size == 1

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

Sequence c(a); // Copy constructor on non-empty sequence

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

assert( c.set(1, "b") == **true**);

assert( c.set(2, "a") == **true**); // Setting tail value

assert( c.remove("a") == 2); // Remove multiple values

assert( c.size() == 1 ); // Check size after removing

Sequence d;

assert( d.insert(0, "First") == 0 ); // Insert when position == size

assert( d.insert(1, "Second") == 1 ); // Insert when position == size

assert( d.insert(2, "Third") == 2 ); // Insert when position == size

assert( d.insert(3, "Fourth") == 3 ); // Insert when position == size

d.swap(d); // Swapping with self

assert( d.size() == 4 ); // Check size after swapping with self

d.swap(a); // Swap with another sequence

assert( d.size() == 3 ); // Check size after swapping with another sequence

assert( a.size() == 4 );

assert( d.find("First") == -1); // Check for new values in swapped sequences

assert( a.find("First") == 0);

assert( d.find("a") == 0 );

assert( d.find("Second") == -1); // Check for new values in swapped sequences

assert( a.find("Second") == 1);

assert( d.find("b") == 1 );

assert( d.set(0, "1") == **true** );

assert( d.set(1, "2") == **true** );

assert( d.set(2, "3") == **true** );

assert( a.find("1") == -1 ); // Check if swapped sequences don't share same address

ItemType willBeOverwritten = "Replace me";

assert( d.get(0, willBeOverwritten) == **true** ); // Get value

assert( willBeOverwritten == "1" );

ItemType wontBeOverwritten = "Unchanged";

assert( d.get(-1, wontBeOverwritten) == **false** ); // Get function on out of bounds position

assert( d.get(4, wontBeOverwritten) == **false** ); // Get function on out of bounds position

assert( wontBeOverwritten == "Unchanged" );

Sequence First;

assert( First.insert(0,"a") == 0 );

assert( First.insert(0,"b") == 0 );

assert( First.insert(0,"c") == 0 );

Sequence Second;

assert( Second.insert(0,"d") == 0 );

assert( Second.insert(0,"e") == 0 );

assert( Second.insert(0,"f") == 0 );

Sequence Result;

concatReverse(First, Second, Result);

assert( Result.size() == First.size() + Second.size() ); // Check result size

assert( Result.find("a") == 0 ); // Check if result sequence is concatenated and reversed

assert( Result.find("b") == 1 );

assert( Result.find("c") == 2 );

assert( Result.find("d") == 3 );

assert( Result.find("e") == 4 );

assert( Result.find("f") == 5 );

Sequence Single;

Sequence SingleResult;

assert( Single.insert(0,"c") == 0 );

assert( Single.insert(0,"a") == 0 );

assert( Single.insert(0,"t") == 0 );

concatReverse(Single, Single, SingleResult); // Call concatReverse with the same sequence

assert( SingleResult.size() == 6); // Check size after concatReverse

assert( SingleResult.find("c") == 0 );

assert( SingleResult.find("a") == 1 );

assert( SingleResult.find("t") == 2 );

assert( SingleResult.remove("c") == 2 ); // Removing after concatReverse on same object

assert( SingleResult.remove("a") == 2 );

assert( SingleResult.remove("t") == 2 );

Sequence EmptyConcat;

Sequence EmptyConcatResult;

concatReverse(EmptyConcat, EmptyConcat, EmptyConcatResult); // concatReverse on empty sequence

assert( EmptyConcatResult.size() == 0 ); // Check size after concatReverse on empty sequence

Sequence NotEmptyConcat;

assert( NotEmptyConcat.insert("A") == 0 );

concatReverse(EmptyConcat, NotEmptyConcat, EmptyConcatResult); // concatReverse on empty sequence and non empty sequence

assert( EmptyConcatResult.size() == 1 ); // Check size

assert( EmptyConcatResult.find("A") == 0 ); // Check original non empty sequence

Sequence largeSequence;

assert( largeSequence.insert(0,"c") == 0);

assert( largeSequence.insert(0,"b") == 0);

assert( largeSequence.insert(0,"a") == 0);

assert( largeSequence.insert(0,"z") == 0);

assert( largeSequence.insert(0,"d") == 0);

assert( largeSequence.insert(0,"c") == 0);

assert( largeSequence.insert(0,"b") == 0);

assert( largeSequence.insert(0,"a") == 0);

Sequence smallerSequence;

assert( smallerSequence.insert(0, "c") == 0 );

assert( smallerSequence.insert(0, "b") == 0 );

assert( smallerSequence.insert(0, "a") == 0 );

assert( subsequence(largeSequence, smallerSequence) == 0 ); // Sequence "abc" is first found in index 0 even though there are 2 "abc"s

assert( subsequence(smallerSequence, largeSequence) == -1 ); // Subsequence is larger than sequence you're looking through

assert( subsequence(largeSequence, largeSequence) == 0 ); // Inputting same sequence

assert( largeSequence.insert(1,"z") == 1);

assert( subsequence(largeSequence, smallerSequence) == 6 ); // Non consecutive subsequence comes before consecutive sequence

Sequence emptySubSequence;

assert( subsequence(largeSequence, emptySubSequence) == -1 ); // Empty subsequence

assert( subsequence(emptySubSequence, emptySubSequence) == -1 ); // Both sequences empty

assert( subsequence(emptySubSequence, smallerSequence) == -1 ); // Empty sequence