Zane Koch

CS32

Smallberg

Project 2 Report

Design**:**

The design of my linked list is the following: a doubly linked list made up of nodes that have a next pointer, prev pointer, and a data value of ItemType pointer to by a single head pointer. There is no tail pointer, or dummy nodes and it is not circular.

A typical sequence might look like the following: int m\_size=4;

|  |
| --- |
| next |
| Prev:  nullptr |
| Data:5 |

|  |
| --- |
| Next:  nullptr |
| Prev |
| Data:60 |

|  |
| --- |
| Next |
| Prev |
| Data:42 |

|  |
| --- |
| next |
| prev |
| Data:6 |

|  |
| --- |
| M\_head |

A typical empty sequence might look like the following: int m\_size=0;

|  |
| --- |
| M\_head:  nullptr |

Pseudo code:

~Sequence:

1. If the list is empty delete m\_head
2. Otherwise:
3. Loop through all nodes using two pointers
4. Delete each node
5. Delete the pointers

Sequence(const Sequence& old) //copy constructor:

1. If the new sequence is not the same as the old:
2. Set m\_head to nullptr
3. Loop each time ‘getting’ value of a node from old and creating a new node with this value using sequence::insert
4. Set m\_size to old m\_size
5. Otherwise do nothing

Assignment operator:

1. Check if this is equal to argument
2. If so return the this object
3. Otherwise set this’s attritubutes to the arguments and return pointer to this

Sequence::insert(int pos, const ItemType value):

1. If the position is not valid return false
2. Otherwise if it’s the first node added, create a node and point head to it
3. Otherwise if its being add to position, create a node and move pointers accordingly
4. Otherwise if its being added to end of list, create a node and point old last node to it
5. Otherwise if its being added to middle of list, create a node and point nodes around it to it

Sequence::insert(const Itemtype& value):

1. Loop over all nodes currently in linked list
2. Get each nodes value
3. Comare this value to value in function argument
4. Insert when find a value <= or if never insert at end

Sequence::erase(int pos):

1. If position is invalid return false
2. If deleting only item, delete first item and set m\_head to null
3. If deleting first item, make temp pointer to second node, delete first node, point m\_head to temp
4. If deleting last node, set second to last node’s next to null and delete last node
5. If deleting in the middle, concatenate node pointers around it and delete node

Sequence::remove:

1. Loop through all nodes while there is still an item with ‘value’
2. Use erase on the location that find returns
3. Increment removed
4. Return removed

Sequence::get

1. Return false if invalid position
2. Iterate through the list to the pos requested
3. Set value to the data at the node requested
4. Return true if made it to that point

Sequence::set

1. Return false if invalid position
2. Loop through every node until get to desired node
3. Using a pointer to this node change its data to desired data
4. Return true

Sequence::find

1. Return -1 if invalid position
2. Check if head points to a node with value, if so return this
3. Otherwise loop through all nodes checking if they have desired data
4. If so return the position of this node
5. Otherwise return -1

Sequence::swap

1. Using the c++ std library swap, swap the data members that a sequence contains
2. Swap size
3. Swap head pointer

Subsequence:

1. Using a loop inside a loop compare every value of the two sequences
2. If you find a value that matches, check if the next value matches
3. If the number of values in a row matches the length of the second sequence return the place in the first sequence this happens
4. If you find a value that matches but the next does not go to next iteration of loop and set the variable that tracks number of matches in a row back to 0
5. If no value that satisfies this is found return -1

Interleave:

1. Check that the result seq is empty
2. If it is not, empty it
3. Find which list is longer
4. Iterate from 0 to the value of the longer list
5. Each iteration, if the value of the iterating variable is less than the length of the first list, insert the value at that position from the first list into result
6. Then check if the value of the iterating variable is less than the length of the second list and if so insert it into the next spot in the loop.
7. Continue doing this until the iterating variable stops being smaller than both at which point the algorithm is complete

Test cases:

//tests for insert

Sequence test;

assert(test.empty() == true);

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

test.insert(0, 20);

test.insert(0, 10);

test.insert(2, 30);

test.insert(2, 25);

test.insert(3, 27);

test.insert(28);

test.insert(1);

test.dump(); //visually check that the values have been inserted properly

cerr << endl;

//tests for erase

assert(test.size() == 7); //check number of vals is== number inserted

test.erase(0);

assert(test.size() == 6);//check decreases by one when erasing

//test.dump();

//cerr << endl;

test.erase(5);

//test.dump();

//cerr << endl;

test.erase(3);

test.dump(); //make sure the correct values were removed

//cerr << endl;

assert(test.size() == 4);//check decrease by two when erasing

Sequence oneitem;

oneitem.insert(10);

//oneitem.dump(); cerr << endl;

oneitem.erase(0);

assert(oneitem.empty() == true); //make sure erase works on empty string

//tests for find

int z;

z=test.find(20);

assert(z == 1);//checks if finds variable in correct location

z = test.find(25);

assert(z == 2);//checks if finds variable in correct location

z = test.find(28);

assert(z == 3);//checks if finds variable in correct location

z = test.find(10);

assert(z == 0);//checks if finds variable in correct location

z = test.find(100);

assert(z == -1);//checks if correctly throws error

z = oneitem.find(10);

assert(z = -1); //checks if correctly throws error

//test for remove

test.dump(); cerr << endl;

test.remove(25);

assert(!test.remove(10000));//makes sure does not remove when value not in list

test.dump();

cerr << endl;

assert(test.size() == 3);//makes sure remove

//test for get

for (int i = 0; i < test.size(); i++)

{

ItemType j;

if (test.get(i, j))

cerr << "true";

else

cerr << "false";

cerr << j << endl;

} //visually check that it ouputs the correct values

//tests for set

test.dump(); cerr << endl;

test.set(0, 2);

assert(!test.set(5, 1000));//make sure does not allow value to be inserted in bad location

test.dump();//visually check value was inserted correctly

cerr << endl;

//tests for swap

Sequence zane;

Sequence a;

zane.insert(0, 1); zane.insert(0, 3); zane.insert(1, 2); zane.insert(3, 0); zane.insert(0, 5);

a.insert(0); a.insert(1); a.insert(2);

cerr << "zane sequence: ";

zane.dump();

cerr << endl << "a sequence: ";

a.dump(); cerr << endl;

zane.swap(a);

cerr << "zane sequence: ";

zane.dump();

cerr << endl << "a sequence: ";

a.dump(); //visually check that the values were swapped correctly

cerr << endl;

zane.remove(0);

zane.dump(); cerr << endl;

a.dump();//make sure editing one seq does not affect the other in anyway

cerr << endl;

//tests for copy constructor

Sequence b; b.insert(0); b.insert(1); b.insert(2);

b.dump(); cerr << endl;

Sequence c = b;

c.dump(); cerr << endl;//visually check that the assignment operator changed values correctly

b.remove(0);

b.dump(); cerr << endl;

c.dump();//make sure that changing on seq does not affect other

//tests for assignment operator

Sequence waba;

waba.insert(1);

waba.insert(10000);

Sequence laba;

laba.insert(0, 24);

laba = waba;

waba.dump(); cerr << endl;

laba.dump(); cerr << endl;//make values get switched

laba.erase(0);

waba.dump(); cerr << endl;

laba.dump(); cerr << endl;//make sure nothing bad happens when changing one string

Sequence nnew;

nnew.insert(1); nnew.insert(1); nnew.insert(1); nnew.insert(1); nnew.insert(1);

nnew.dump(); cerr << endl;

Sequence ham;

ham.insert(5);

ham.dump(); cerr << endl;

nnew = ham;

nnew.dump(); cerr << endl;

ham.dump();

//tets for subsequence

Sequence seq1; //30 21 63 42 17 63 17 29 8 32

seq1.insert(0,30); seq1.insert(1,21); seq1.insert(2,63); seq1.insert(3,42); seq1.insert(4,17); seq1.insert(5,63); seq1.insert(6,17); seq1.insert(7,29); seq1.insert(8,8); seq1.insert(9,32);

Sequence seq2;

seq2.insert(0,63); seq2.insert(1,17); seq2.insert(2,29);

assert(subsequence(seq1, seq2) == 5);//check that returns correct value

Sequence seq3;

assert(subsequence(seq1, seq3) == -1);//check that returns -1 for empty second seq

//tests for interleave

Sequence seq11; //30 21 63 42 17 63

seq11.insert(0, 30); seq11.insert(1, 21); seq11.insert(2, 63); seq11.insert(3, 42); seq11.insert(4, 17); seq11.insert(5, 63);

Sequence seq22; // 42 63 84 19

seq22.insert(0, 42); seq22.insert(1, 63); seq22.insert(2, 84); seq22.insert(3, 19);

seq11.dump(); cerr << endl;

seq22.dump(); cerr << endl;

Sequence result;

result.insert(1);

interleave(seq11, seq22, result);//with empty return

interleave(seq22, seq11, result);//with non-empty return

result.dump(); cerr << endl;//visually check that correctly changed return seq