Preston Futaba

Professor Smallberg

Computer Science 32

31 January 2023

Project 2 Report

Implementation of doubly-linked list:

For my doubly-linked list, no dummy nodes were used-thus it is not circular. Instead I sandwiched the nodes containing items between 2 nullptrs, so the pointer to the previous node of the head node is nullptr and the next node of the tail node is also nullptr. This defines head and tail pointer as nullptrs. Each node contains information on its own data (m\_item) and pointers to the previous and next nodes. Also, the lists are not sorted.

Pseudocode:

**Set insert:**

First check if the set already contains the value that is trying to be inserted:

If so, exit and return false (aka the function failed)

If the set is empty:

Create a new node

Give the new node’s item the value parameter

Sandwich new node with both nullptrs because it’s the only node

Set’s head and tail points to new node

Increment set size

If set isn’t empty:

Create a new node

Give the new node’s item the value parameter

New node is attached to end of list:

New node’s previous pointer points to set’s current tail

New node’s next pointer points to the null pointer

Point set’s tail to this new node (making it the last node)

Increment set size

Return true, aka insertion successful

**Set erase:**

First check if the set contains the value that is trying to be erased:

If not contained, exit and return (false) function failed

Find node that contains the value (m\_item)

Point previous to next and next to previous

Delete node containing value

**Set get:**

Check if pos is a valid value and is less than set size

Check each node’s value to all the other nodes

If that current node’s value is greater

Increment counter

If that item is greater than strictly pos items

Give that m\_item to value

Return true for successful get

Return false if no successful gets after iteration

Else

Return false if pos is not valid

**Set swap:**

Swap set size in this with other

Swap head in this with other

Swap tail in this with other

**Unite:**

//Was originally going to do a bunch of comparisons and fetch each node’s value, but why do that when my insert function does that for me? Aka more elegant

Copy s1 into result

For each value in s2:

Try to insert it into result

//Insert function means no repeats

**butNot:**

Check if sets are the same

subtracting them results in an empty set

Copy contents of s1 into result

For each value in s2:

Try to erase it from result

Set food; // default constructor

// Empty set:

assert(food.size() == 0); // empty set has size 0

assert(food.empty()); // empty set is empty

assert(!food.erase("apple")); // can't remove from empty set

assert(food.insert("apple")); // insert into empty set

assert(food.size() == 1); // did size increase from 0 to 1 after inserting "apple"

assert(!food.empty()); // not empty anymore, has 1 item

assert(food.erase("apple")); // erase only 1 element in set

// Non empty set:

assert(food.insert("apple")); // reinsert 1 item (arbitrary as just 1 item)

assert(food.insert("banana")); // add 2nd item

assert(food.size() == 2); // size = 2 items, apple + banana

assert(food.contains("banana")); // test contains fn

ItemType x = "steak"; //give x an arbitrary value so we can test get fn

assert(food.get(0, x) && x == "apple"); // getting object 0 (greater than 0 objects)

assert(food.get(1, x) && x == "banana"); // getting object 1 (greater than 1 objects)

assert(!food.insert("apple")); // can't insert duplicate

assert(food.insert("cucumber")); // add 3rd element

Set food2(food); // test cc

assert(food2.get(0, x) && x == "apple"); // getting object 0 (greater than 0 objects)

assert(food2.get(1, x) && x == "banana"); // getting object 1 (greater than 1 objects)

assert(food2.get(2, x) && x == "cucumber"); // getting object 2 (greater than 2 objects)

assert(food2.insert("durian")); // insertion testing

assert(food2.erase("apple")); // test removing works

assert(!food.contains("durian")); // make sure food2 and food are not connected to same list

assert(food.contains("apple")); // make sure food2 and food are not connected to same list

Set food3;

food3 = food; // test AO

assert(food3.get(0, x) && x == "apple"); // getting object 0 (greater than 0 objects)

assert(food3.get(1, x) && x == "banana"); // getting object 1 (greater than 1 objects)

assert(food3.get(2, x) && x == "cucumber"); // getting object 2 (greater than 2 objects)

assert(food3.erase("apple")); // test remove

assert(food.contains("apple")); // make sure food3 and food are not connected to same list

unite(food, food2, food); // check if unite works, food unites into food

// checking if unite worked correctly

assert(food.get(0, x) && x == "apple"); // getting object 0 (greater than 0 objects)

assert(food.get(1, x) && x == "banana"); // getting object 1 (greater than 1 objects)

assert(food.get(2, x) && x == "cucumber"); // getting object 2 (greater than 2 objects)

assert(food.get(3, x) && x == "durian"); // getting object 3 (greater than 3 objects)

butNot(food, food2, food); //check if butNot works, food subtracted to food

// did subtract work?

assert(food.get(0, x) && x == "apple"); // getting object 0 (greater than 0 objects)

assert(food.size()==1); // only apple in food after butNot

food.swap(food2); // does swap work properly

assert(food.get(0, x) && x == "banana"); // food has food2's contents

assert(food.get(1, x) && x == "cucumber"); // food has food2's contents

assert(food2.get(0, x) && x == "apple"); // food2 has food's contents