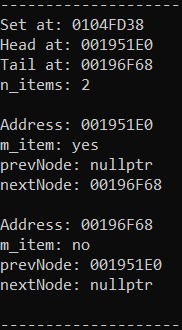
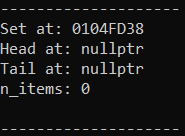
# Description of Doubly Linked List

For the doubly linked list used by my set, no dummy nodes were used. Instead, the pointer to the previous node of the head node is nullptr and the next node of the tail node is also nullptr. In addition, the head and tail pointer stored in the set would both be nullptr. There is no specific ordering to the list and it is not circularly linked. Each node contains information on its own data and pointers to the previous and next nodes. The set does contain information on the head and tail nodes.

Examples of an empty set and a set with elements in it are illustrated in the following pictures

# Pseudocode

Set constructor:

Set the number of items to zero

Point the head and tail to null because there are no nodes

Set destructor:

If there are no items in the list:

There is no allocated memory, so no need to do anything

Otherwise:

Traverse through each node:

Delete this node

Jump to the next node

Set copy constructor:

Set the number of items to other’s number of items

Traversing through each node in other:

Make a new node in this set

Set its data to the data from the respective node in other

Connect this node to the previous and next node

Set equal operator override:

Make sure this object isn’t being set to itself:

If not, copy the right hand side to a temp variable

Swap the left hand side with that temp variable

Return itself

Set insert:

Check if this set contains the value to be inserted:

If it does, exit and return that this function failed

If this set is empty:

Create a new node

Fill its data with the given value

Connect it to the null pointer on either side because it’s the only node

Point the set’s head and tail to it

Increment number of items

If this set isn’t empty:

Create a new node

Fill its data with the given value

Attach it to the end:

Connect its previous pointer to the set’s current tail

Connect its next pointer to the null pointer

Point the set’s tail to this new node

Increment number of items

Return that the insertion was successful

Set erase:

Check if this set contains the value to be erased:

If it doesn’t, exit and return that this function failed

Find the node that contains the value

Point the previous node to the next node and next to the previous

Delete the node containing the value

Set get:

Check if given index is value:

If it isn’t, return that this function failed

Create a temporary copy of this set:

Delete the minimum value from the copy pos number of times

Set value to the minimum value remaining in the copy, which will be the desired value

Set swap:

Swap the number of items in this with other

Swap the head in this with other

Swap the tail in this with other

Unite:

Copy the contents of s1 into result

For each value in s2:

Try to insert it into result

Subtract:

Copy the contents of s1 into result

For each value in s2:

Try to remove it from result

# Test cases:

The tests were performed on a set of strings (i.e. ItemType was a type alias for std::string).

The tests were additionally performed using arbitrary unsigned longs to make sure the type aliasing was done correctly.

// default constructor

Set ss;

// For an empty set:

assert(ss.size() == 0); // test size on empty set

assert(ss.empty()); // test empty

assert(!ss.erase("roti")); // test a failure to remove on an empty set

assert(ss.insert("roti")); // test insertion into empty set

assert(ss.size() == 1); // test if size is 1 after inserting 1 item

assert(!ss.empty()); // test if empty works on a set of size 1

assert(ss.erase("roti")); // test if erase works with only 1 element

// same as first 3 tests to make sure roti was removed correctly

assert(ss.size() == 0); // test size on empty set

assert(ss.empty()); // test empty

assert(!ss.erase("roti")); // test a failure to remove on an empty set

assert(ss.insert("roti")); // test reinserting 1 item

assert(ss.insert("pita")); // test adding in 2nd item

assert(ss.size() == 2); // test that size is working correctly with 2 items

assert(ss.contains("pita")); // make sure that contains works properly

ItemType x = "laobing";

assert(ss.get(0, x) && x == "pita"); // make sure get works when getting object 0

assert(ss.get(1, x) && x == "roti"); // make sure get works when getting object 1

assert(!ss.insert("roti")); // make sure it doesn’t allow inserting something already in it

assert(ss.insert("naan")); // add third element to set

Set ss2(ss); // checking copy constructor

assert(ss2.get(0, x) && x == "naan"); // make sure get works when getting object 0

assert(ss2.get(1, x) && x == "pita"); // make sure get works when getting object 1

assert(ss2.get(2, x) && x == "roti"); // make sure get works when getting object 2

assert(ss2.insert("white bread")); // make sure insertion works

assert(ss2.erase("naan")); // check if removing works

assert(!ss.contains("white bread")); // make sure ss2 and ss are not connected to the same linked list

assert(ss.contains("naan")); // make sure ss2 and ss are not connected to the same linked list

Set ss3;

ss3 = ss; // make sure operator= override works

assert(ss3.get(0, x) && x == "naan"); // make sure get works when getting object 0

assert(ss3.get(1, x) && x == "pita"); // make sure get works when getting object 1

assert(ss3.get(2, x) && x == "roti"); // make sure get works when getting object 2

assert(ss3.erase("naan")); // check if removing works

assert(ss.contains("naan")); // make sure ss3 and ss are not connected to the same linked list

unite(ss, ss2, ss); // check if unite works, including if ss1 unites into ss1

// checking if unite worked correctly

assert(ss.get(0, x) && x == "naan"); // make sure get works when getting object 0

assert(ss.get(1, x) && x == "pita"); // make sure get works when getting object 1

assert(ss.get(2, x) && x == "roti"); // make sure get works when getting object 2

assert(ss.get(3, x) && x == "white bread"); // make sure get works when getting object 2

subtract(ss, ss2, ss); //check if subtract works, including if ss1 is subtracted from and stored to

// checking if subtract worked correctly

assert(ss.get(0, x) && x == "naan"); // make sure get works when getting object 0

assert(ss.get(1, x) && x == "white bread"); // make sure get works when getting object 2

ss.swap(ss2); // check if swap works properly

assert(ss.get(0, x) && x == "pita"); // checking that ss now has what ss2 had

assert(ss.get(1, x) && x == "roti"); // checking that ss now has what ss2 had

assert(ss2.get(0, x) && x == "naan"); // checking that ss2 now has what ss had

assert(ss2.get(1, x) && x == "white bread"); // checking that ss2 now has what ss had