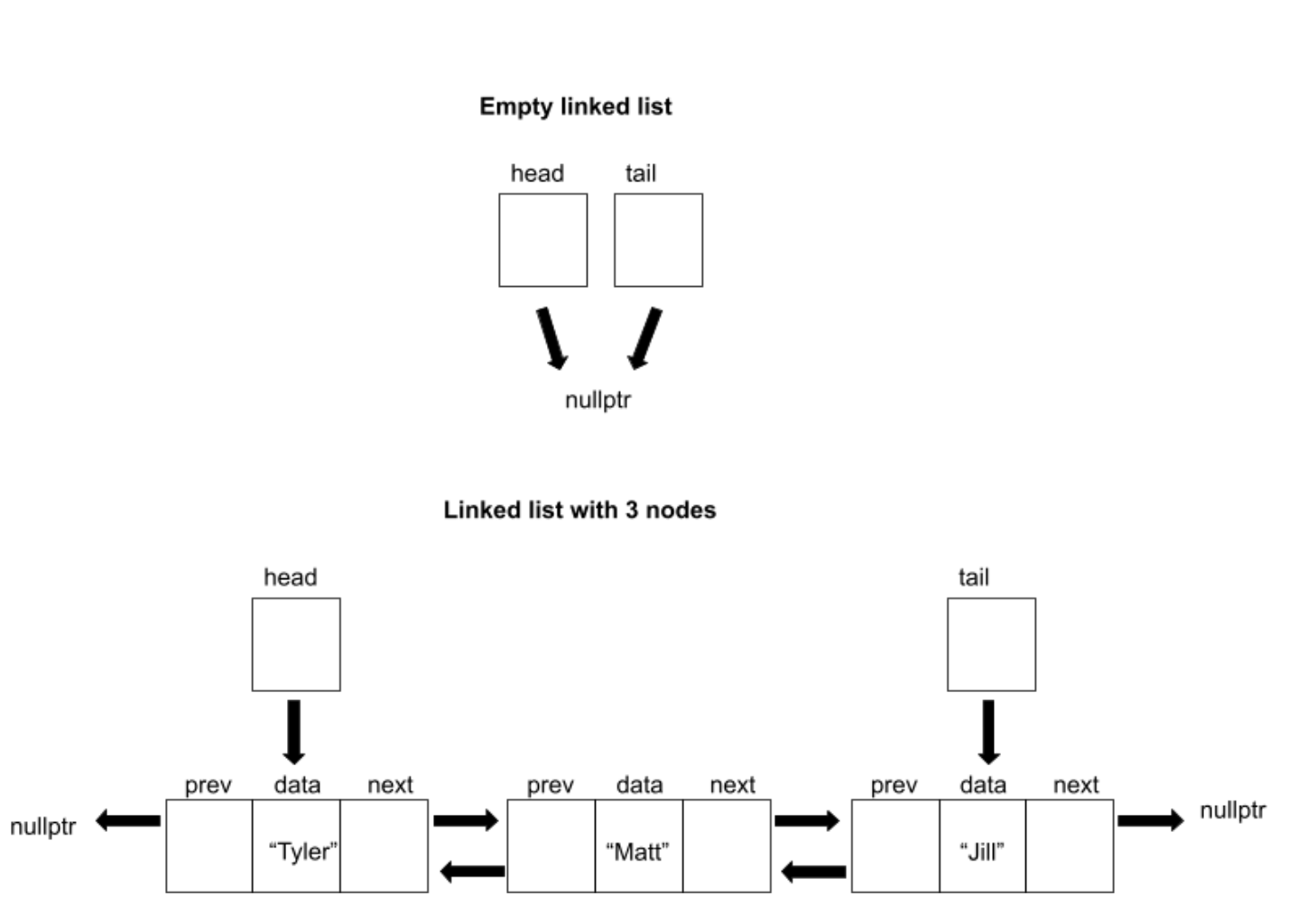
Tyler Stovsky

705512370

Design description:

My doubly linked list design is not circular nor does it have a dummy node. Every node has an ItemType named data as well as a pointer to the next node and a pointer to the previous node. There is a head node in which its pointer to the previous node is nullptr, and a tail node in which its pointer to the next node is nullptr. When an empty set is created, both the head and tail point to nullptr. When inserting an entry into the list, the entry gets added to the end of the list. Therefore, the list is not sorted in a specific way. Here are some examples:

Pseudocode:

**Set::Set()**

Set size to 0

Set head and tail to null pointer

**Set::~Set()**

Loop through each node:  
 set current pointer to the head node

set head node to head’s next node

deallocate current pointer

**Set::Set(const Set& other)**

Set size to other’s size

If the size is 0:

set head and tail to nullptr

Otherwise:

create head node  
 Loop through every node in other:  
 create a new node

set data to other’s data

connect node to previous and next node

connect tail node

**Set& Set::operator=(const Set &s)**

Check if object is being assigned to itself

If not, use copy constructor to copy right side into a temporary Set

use swap function to swap left side with the temporary Set

Return itself

**bool Set::insert(const ItemType &value)**

Check if the value is already in set:

If so, return false

Create a new node

Set new node’s data to value

If the list is empty:

Point head and tail to the new node

Connect new node’s previous and next pointer to null pointer

Otherwise:

Connect tail’s next pointer to new node

Connect new node’s previous pointer to tail

Point tail to new node

Connect new node’s next pointer to null pointer

Increment size of the list

**bool Set::erase(const ItemType &value)**

Check if the value is already in the set

If not, return false

Find the node to be deleted

If the node isn’t the head node:

connect previous node to node’s next node

Otherwise:

connect head node to next entry

If the node isn’t the tail node:  
 connect next node to node’s previous node

Otherwise:  
 connect tail node to previous entry

Decrease the size of the list

Delete the node

**bool Set::contains(const ItemType &value) const**

Check if the list is empty:

If so, return false

Loop through each node:

If a node contains the data:

return true

**bool Set::get(int i, ItemType &value) const**

Check if the index is valid:

If not, return false

Create a temporary copy of our Set

Loop the index number of times:

Find the maximum value and delete it

Find the new maximum

Set value equal to the new maximum

**void Set::swap(Set &other)**

Swap the sizes of the lists

Swap the head nodes

Swap the tail nodes

**void unite(const Set& s1, const Set& s2, Set& result)**

Set result equal to s1

Loop through the items in s2:  
 Try to insert each item into the result

**void difference(const Set& s1, const Set& s2, Set& result)**

Set result equal to s1

Loop through the items in s2:

If the result contains the item:

Delete the item

Otherwise:

Insert the item

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

Test cases:

Set s; // default constructor creates empty set

assert(s.size() == 0); // size should be 0 when set is created

assert(s.empty()); // set should be empty when created

assert(!s.erase("Tyler")); // cannot erase from an empty set

assert(s.insert("Tyler")); // insert into an empty set

assert(s.size() == 1); // size should increase after insertion

assert(!s.empty()); // set should not be empty after insertion

assert(s.erase("Tyler")); // test erase when there is only one item

assert(s.size() == 0); // size should decrease after deletion

assert(s.insert("Lexi")); // reinsert after a deletion of every item

assert(s.insert("Matt")); // insert a second item

assert(s.insert("Jill")); // insert a third item

assert(!s.insert("Matt")); // can't insert the same item twice

assert(s.size() == 3); // size should work even after a failed insertion

assert(s.erase("Lexi")); // erase the first item with other items in the list

ItemType x;

assert(s.get(0, x) && x == "Matt"); // function should get largest item

assert(s.get(1,x) && x == "Jill"); // function should get second largest item

assert(s.erase("Jill")); // erase the last item

assert(!s.get(99, x)); // get function index too large

Set s1(s); // test the copy constructor

assert(s1.get(0,x) && x == "Matt"); // items in the list should be the same

assert(s1.insert("Mike")); // insert an item to a copied set

assert(s1.erase("Matt")); // erase an item from a copied set

assert(!s.contains("Mike")); // set we copied from shouldn't be connected to the new set

assert(s.contains("Matt")); // set we copied from shouldn't be connected to the new set

Set s2;

s2 = s; // test assignment operator

assert(s2.get(0,x) && x == "Matt"); // items in the list should be the same

assert(s2.insert("Mike")); // insert item into the assigned set

assert(s2.erase("Matt")); // erase an item from the assigned set

assert(!s.contains("Mike")); // set we copied from shouldn't be connected to the new set

assert(s.contains("Matt")); // set we copied from shouldn't be connected to the new set

Set u1, u2, u3;

assert(u1.insert("Pizza"));

assert(u1.insert("Dough"));

assert(u2.insert("Dough"));

assert(u2.insert("Pasta"));

unite(u1,u2,u3); // unite when result set is empty

assert(u3.get(0, x) && x == "Pizza"); // greatest item is "Pizza"

assert(u3.get(1, x) && x == "Pasta"); // second greatest item is "Pasta"

assert(u3.get(2, x) && x == "Dough"); // third greatest item is "Dough"

assert(u3.size() == 3); // "Dough" should only be in the list once

unite(u3,u2,u1); // unite when result set has items in it

assert(u1.get(0, x) && x == "Pizza"); // greatest item is "Pizza"

assert(u1.get(1, x) && x == "Pasta"); // second greatest item is "Pasta"

assert(u1.get(2, x) && x == "Dough"); // third greatest item is "Dough"

assert(u1.erase("Pasta")); // revert u1 back to its original

difference(u1, u2, u3); // difference when an item is in common

assert(u3.get(0, x) && x == "Pizza"); // greatest item is "Pizza"

assert(u3.get(1, x) && x == "Pasta"); // second greatest item is "Pasta"

assert(!u3.contains("Dough")); // "Dough" appears in both so it shouldn't be included

assert(u2.erase("Dough")); // remove common item from u1 and u2

unite(u1, u2, u3); // unite when the two sets have no common items

assert(u3.get(0, x) && x == "Pizza"); // greatest item is "Pizza"

assert(u3.get(1, x) && x == "Pasta"); // second greatest item is "Pasta"

assert(u3.get(2, x) && x == "Dough"); // third greatest item is "Dough"

difference(u1, u2, u3); // difference when no items are in common

assert(u3.get(0, x) && x == "Pizza"); // greatest item is "Pizza"

assert(u3.get(1, x) && x == "Pasta"); // second greatest item is "Pasta"

assert(u3.get(2, x) && x == "Dough"); // third greatest item is “Dough"

Set swap1, swap2;

assert(swap1.insert("Tosh"));

assert(swap1.insert("Eli"));

assert(swap2.insert("Chris"));

assert(swap2.insert("Andrew"));

assert(swap1.get(0, x) && x == "Tosh"); // "Tosh" is largest value before swap

assert(swap1.get(1, x) && x == "Eli"); // "Eli" is second largest value before swap

assert(swap2.get(0, x) && x == "Chris"); // "Chris" is largest value before swap

assert(swap2.get(1, x) && x == "Andrew"); // "Andrew" is second largest value before swap

swap1.swap(swap2); // test the swap function

assert(swap1.get(0, x) && x == "Chris"); // "Chris" is largest value after swap

assert(swap1.get(1, x) && x == "Andrew"); // "Andrew" is second largest value after swap

assert(swap2.get(0, x) && x == "Tosh"); // "Tosh" is largest value after swap

assert(swap2.get(1, x) && x == "Eli"); // "Eli" is second largest value after swap