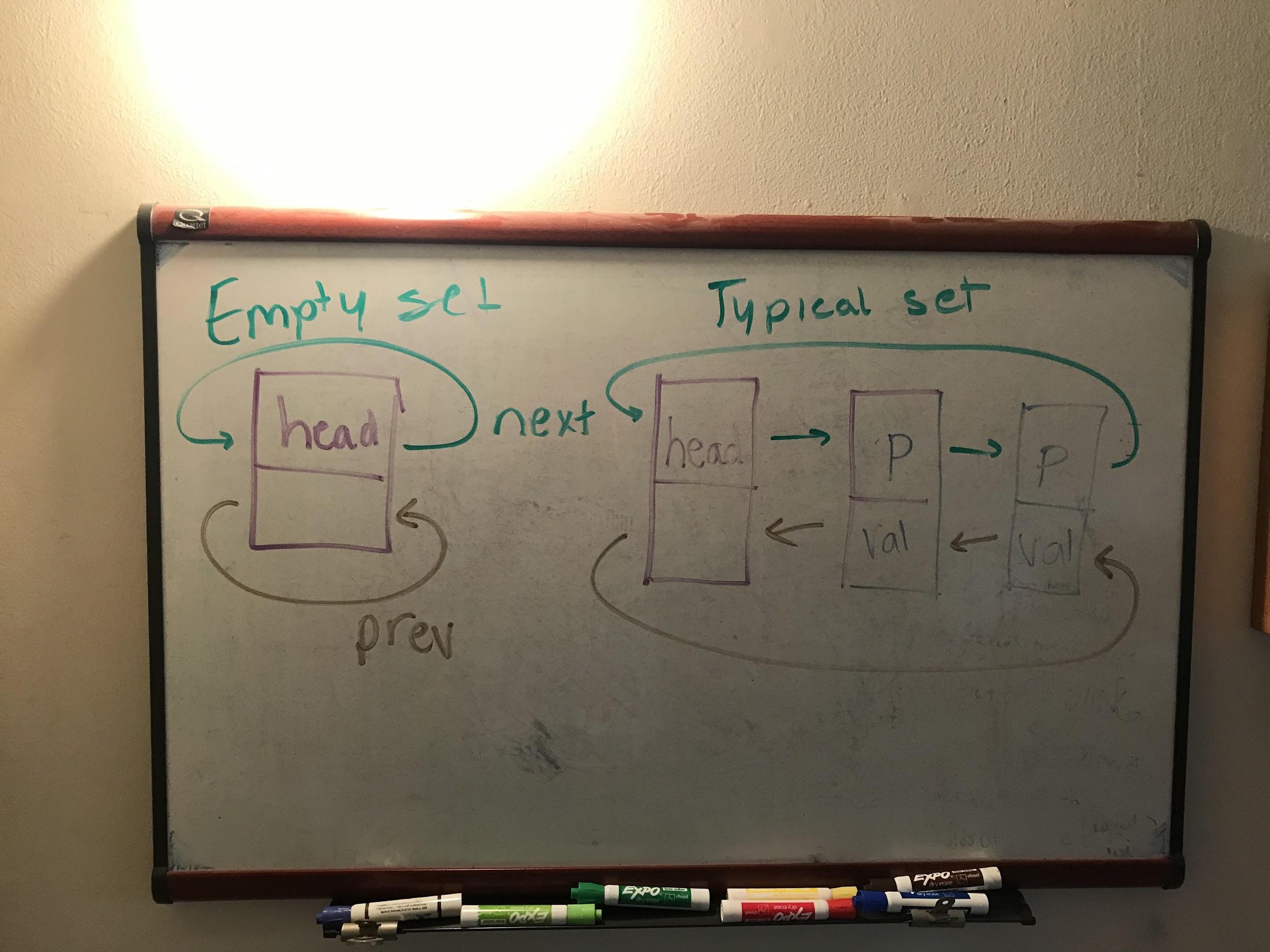
For my doubly-linked list implementation, I implemented a doubly-linked circular list with a dummy node, where I did not use any nullptr. My head was a dummy node that pointed to itself (both its next and previous) in an empty set. In a non-empty set, the last node’s next pointed directly to head and the head’s previous pointed to the last node. This way, I did not have to check for many special corner cases like a 1 node set. The list is ordered in a way from least to greatest, where the largest value is placed at the end of the list (the last node) and the smallest value is placed at the first node (the node after the head)



Pseudocode for non-trivial functions

Set constructor : sets size to 0 // constructor

{

Makes new Node for head

Head next and previous point to itself

}

Set copy constructor : set size to other size

{

if (other head next and previous point to itself) // if the set is empty

{

Makes new Node for head

Head next and previous point to itself

}

else // non empty list

{

Makes new Node for head

Node Current starts at head

Node otherCurrent starts at the first node after head

while (otherCurrent is not head)

{

Makes new Node for the node after Current

The Node after current’s value is the same as otherCurrent’s value (since otherCurrent is the node after head just like Current next is)

Points the previous arrow of the node after Current to Current

The node after the node that Current points to is head

Moves Current to the next Node

Moves otherCurrent to the next Node

}

Head previous points to the last Node (which is Current when the loop is over)

}

}

Set assignment operator

{

if (the left hand side is not equal to the right hand side of the equal sign)

{

Makes a temp Set copy of the right hand side

Swaps temp

}

Returns this

}

Set insert

{

if the value is already in the set

{

Don't insert it

}

// if not in the set

Creates new Node pointer

Sets the new node val to value

if (head next and previous points to head, the set is empty)

{

New node points to head

New node prev points to head

Head points to p

Head prev points to p

Increase size by one

Insert returns true

}

Makes new Node that starts at the first node after head

while (now is not head)

{

if (the desired insert value is less than the current node’s value

{

the new node that has value points to the current node

the new node's previous arrow points to where the current node's previous was pointing

the node before the new node now points to the new node

the previous of the node after the new node points back to the new node

Increase size

Insert is true

}

now = now->next; // keeps traversing through the list

}

if (the desired insert value is greater than the last node’s value and all previous elements)

{

New node points to head (it will be placed at the end of the list)

New node's previous arrow points to where head previous was previously pointing (the last node)

sets the original last node's next to the new node

Points head previous to the new last node

Increase size

Insert is true

}

return false;

}

Set erase

{

if ( if it is empty, head points to itself)

{

Don't erase anything

}

if (it doesn't contain the desired value to erase)

{

Don't erase anything

}

else

{

Sets node p to the first node after head

while (p is not head)

{

if (the desired erase value is the value at p)

{

Break out of loop (since p now has the desired value to erase)

}

Traverse through the list with p next

}

Make node getRekt and point it to p

Sets the previous's node's next to where the to be deleted node is pointing next

Sets the next node's previous to where the to be deleted node is pointing behind it

deletes the node getRekt

Decreases the size

Erase is true

}

}

Set contains

{

Makes node p first node after head

while (p is not head)

{

if (value at p is the desired value)

{

Set contains the value;

}

Traverses through list with p next

}

If it makes it out of the loop without ever meeting the if condition, it does not contain the value

}

Set get

{

Makes node p, points it to the first node after head

Starts a counter at 0

while (p is not head)

{

if (count is equal to the desired position) (since the list is already sorted, the number of items the value is greater than is the same as its position in the list)

{

The value of the node p is copied into value

The get is achieved (True)

}

Traverses through list with p next;

Increase the count for every node that is checked

}

If it makes out of the list (due to the desired position being larger than the size of the list or negative), return false

}

Set swap

{

Swap the heads using temp

Swap the sizes using a temp

}

unite

{

Makes temp = s1 in case of aliasing

for (starts i at 0; i less than the size of the second set; increment i)

{

Make uninitialized ItemType to copy the value into using get

Gets the nth position that the value is greater than (using get), copies it into x

Inserts x into the temp (which is s1), if s1 already contains it, it wont get inserted since it is checked in insert

}

Sets result to temp (which is s1 with elements of s2 added)

}

Subtract

{

Makes temp = s1 in case of aliasing

for (starts i at 0; i less than the size of the second set; increment i)

{

Make uninitialized ItemType to copy the value into using get

Gets the nth position that the value is greater than (using get), copies it into x

Erases x from the temp (which is s1) (since erase checks if the set does not contain the value, it will only erase once)

}

Sets result to temp (which is s1 with elements of s2 added)

}

Set destructor

{

Makes node p, points it to head->next

while (p is not head)

{

Makes node n equal to the node after p (2 nodes after head)

Deletes p, the node prior to n

Points p to where n is pointing

}

delete head

}

Test cases

Set s69; // tests insert, adding inserts to the end

s69.insert("12");

s69.insert("24");

s69.dump();

s69.insert("");

s69.dump();

Set s;

assert(s.empty()); // tests constructor and empty

assert(s.size() == 0); // tests size and empty

s.insert("tati"); // tests insert, seeing if it does not insert if the value is already there

s.insert("tati");

s.insert("roti");

s.insert("sing");

assert(!s.empty());

s.dump();

assert(s.size() == 3);

s.erase("roti"); // tests erase

s.dump();

s.erase("sing");

s.dump();

s.erase("tati");

s.dump();

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

Set ss4; // ItemType is std::string

ss4.insert("ccc"); // tests get

ss4.insert("aaa");

ss4.insert("bbb");

ss4.dump();

ItemType x = "xxx";

assert(!ss4.get(3, x) && x == "xxx"); // x is unchanged

assert(ss4.get(1, x) && x == "bbb"); // "bbb" is greater than

const ItemType DUMMY\_VALUE = "hello";

const ItemType V1 = "abc";

Set sss;

assert(sss.empty());

ItemType d = DUMMY\_VALUE;

assert(!sss.get(0, d) && d == DUMMY\_VALUE); // v unchanged by get failure

sss.insert(V1);

assert(sss.size() == 1);

assert(sss.get(0, d) && d == V1);

Set ss; // testing insert (insert works for inserting into an empty set, 1 element set, typical set) and erase

assert(ss.empty());

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

ss.insert("lavash");

ss.insert("roti");

ss.insert("chapati");

ss.insert("injera");

ss.insert("roti");

ss.insert("matzo");

ss.insert("injera");

ss.insert("");

ss.insert("");

assert(ss.size() == 6);

assert(!ss.empty());

ss.erase("injera");

assert(ss.size() == 5);

ss.erase("");

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

Set st; // testing get

st.insert("lavash");

st.insert("roti");

st.insert("chapati");

st.insert("injera");

st.insert("roti");

st.insert("matzo");

st.insert("injera");

assert(st.size() == 5); // duplicate "roti" and "injera" were not added

st.get(0, x);

assert(x == "chapati"); // "chapati" is greater than exactly 0 items in ss

st.get(4, x);

assert(x == "roti"); // "roti" is greater than exactly 4 items in ss

st.get(2, x);

assert(x == "lavash"); // "lavash" is greater than exactly 2 items in ss

Set s2; // testing contains and size, that insert properly updates the size

s2.insert("dosa");

assert(!s2.contains(""));

s2.insert("tortilla");

s2.insert("");

s2.insert("focaccia");

assert(s2.contains(""));

s2.erase("dosa");

assert(s2.size() == 3 && s2.contains("focaccia") && s2.contains("tortilla") && s2.contains("") && !s2.contains("dosa"));

std::string v;

assert(s2.get(1, v) && v == "focaccia");

assert(s2.get(0, v) && v == "");

Set n; // testing get

n.insert("");

n.insert("1");

n.insert("2");

n.insert("3");

n.insert("4");

std::string num;

n.get(4, num);

assert(num == "4");

n.get(3, num);

assert(num == "3");

n.get(2, num);

assert(num == "2");

n.get(1, num);

assert(num == "1");

n.get(0, num);

assert(num == "");

Set ss1;

ss1.insert("laobing");

Set ss2;

ss2.insert("matzo"); // testing swap

ss2.insert("pita");

ss1.swap(ss2);

assert(ss1.size() == 2 && ss1.contains("matzo") && ss1.contains("pita") &&

ss2.size() == 1 && ss2.contains("laobing"));

Set set;

set.insert("dingding");

set.insert("singsong");

set.insert("linglong");

set.insert("wingwong");

set.insert("bingbong");

set.erase("bingbong");

s2.swap(set);

assert(set.contains("") && set.contains("focaccia") && set.contains("tortilla") && s2.contains("dingding") && s2.contains("singsong") && !s2.contains("bingbong"));

Set a;

// test empty, size

assert(a.empty());

a.insert("c");

a.dump();

assert(!a.empty());

assert(a.size()==1);

assert(a.contains("c"));

assert(!a.contains("b"));

// test contains

a.insert("b");

a.insert("a");

a.dump();

assert(a.contains("c"));

assert(a.contains("b"));

assert(a.contains("a"));

assert(!a.contains("d"));

// test insert

a.insert("d");

a.insert("d");

a.insert("h");

a.insert("i");

a.insert("f");

a.insert("e");

a.insert("g");

//a.dump();

Set b;

// test erase with only one node

b.insert("a");

//b.dump();

b.erase("a");

//b.dump();

assert(!b.contains("a"));

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

// test erase first node

Set c;

c.insert("a");

c.insert("b");

c.erase("a");

//c.dump();

assert(!c.contains("a"));

assert(c.size()==1);

// test erase last node

Set z;

z.insert("a");

z.insert("b");

z.erase("b");

//d.dump();

assert(!z.contains("b"));

assert(z.size()==1);

// test erase middle node

Set e;

e.insert("c");

e.insert("a");

e.insert("b");

e.erase("b");

e.erase("a");

e.erase("b");

e.dump();

assert(!e.contains("b"));

assert(e.size()==1);

// test get

Set f;

f.insert("ccc");

f.insert("aaa");

f.insert("bbb");

x = "xxx";

assert(!f.get(3, x) && x == "xxx"); // x is unchanged

assert(f.get(1, x) && x == "bbb"); // "bbb" is greater than exactly 1 item

assert(f.get(2, x) && x == "ccc"); // "bbb" is greater than exactly 2 item

assert(f.get(0, x) && x == "aaa"); // "bbb" is greater than exactly 0 item

//test copy constructor

//empty set

Set g;

Set h = g;

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

//typical set

Set i;

i.insert("ccc");

i.insert("aaa");

i.insert("bbb");

Set j = i;

assert(j.size()==3);

j.dump();

Set k;

k.insert("a");

Set l = k;

l.dump();

assert(l.size()==1);

//test swap

Set m;

Set n2;

m.insert("a");

m.insert("b");

m.insert("c");

n2.insert("d");

n2.insert("e");

m.swap(n2);

assert(m.size()==2);

assert(n2.size()==3);

m.dump();

n2.dump();

//swap empty set with one node set

Set o;

Set p;

o.insert("d");

o.swap(p);

assert(p.size()==1);

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

o.dump();

p.dump();

//test unite

Set q; // 2 3 5 8 9

Set r; // 3 5 6 8 10

Set s7; // 1 2 4 8 7 11

q.insert("2");

q.insert("8");

q.insert("3");

q.insert("9");

q.insert("5");

r.insert("6");

r.insert("3");

r.insert("8");

r.insert("5");

r.insert("10");

s7.insert("1");

s7.insert("4");

s7.insert("2");

s7.insert("7");

s7.insert("11");

s7.insert("8");

s7.dump();

unite(q, r, s7);

s7.dump();

subtract(q, r, q); // test that it works with aliasing

q.dump();

Set abc;

Set def;

Set ghk;

unite(abc, def, ghk);

ghk.dump(); // test unite on empty sets

abc.insert("1");

abc.insert("2");

def.insert("3");

ghk.insert("4");

ghk.insert("5");

ghk.erase("5");

ghk.erase("4");

ghk.dump();

unite(abc, def, ghk);

ghk.dump();

// uniting empty lists and one node lists

Set t;

Set u;

Set v2;

t.insert("6");

v2.insert("5");

unite(t, u, v2);

// t.dump();

// u.dump();

// v.dump();

// same set

Set z2;

Set y;

z2.insert("6");

// z.dump();

y.insert("5");

unite(z2, y, z2);

// z.dump();

// y.dump();

// test subtract

Set one; // 1 2 3

Set two; // 4 2 3

Set three;

one.insert("1");

one.insert("2");

one.insert("3");

two.insert("4");

two.insert("2");

two.insert("3");

subtract(one, two, three);

three.dump(); // 1

// testing empty second set

Set six; //

Set four; // 4 2 3

Set five;

four.insert("4");

four.insert("2");

four.insert("3");

subtract(six, four, five);

//five.dump(); //

//testing empty first set

Set seven;

Set eight;

Set nine;

seven.insert("4");

seven.insert("2");

seven.insert("3");

subtract(seven, eight, nine);

//nine.dump();

// result contains value

Set ten;

Set eleven;

Set twelve;

ten.insert("4");

ten.insert("2");

ten.insert("3");

eleven.insert("3");

twelve.insert("3");

subtract(ten, eleven, twelve);

twelve.dump();

//name aliasing

Set X;

Set Y;

Set Z;

X.insert("10");

X.insert("30");

X.insert("40");

X.insert("50");

Y.insert("10");

Y.insert("20");

Y.insert("40");

Y.insert("60");

Y.dump();

unite(X, Y, X);

X.dump();

X.erase("20");

X.erase("60");

subtract(X, Y, X);

X.dump();

//testing insert with inserting node to the end of the list

Set okay;

okay.insert("1");

okay.insert("2");

okay.dump();

Set okay2;

okay2.insert("1");

okay2.insert("2");

subtract(okay, okay2, Z);

Z.dump();

Set him;

him.insert("1");

him.insert("2");

him.insert("3");

him.insert("69");

him.insert("5");

him.insert("4");

assert(him.contains("1") && him.contains("2") && him.contains("69") && him.size() == 6);

him.erase("1");

assert(!him.contains("1") && him.size() == 5);

him.dump();

him.erase("69");

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

him.dump();

him.insert("");

him.dump();

//testing that get fails when the desired position is out of range of the set

Set that;

that.insert("");

that.insert("peen");

assert(that.get(0, x) && x == "");

assert(that.get(1, x) && x == "peen");

assert(!that.get(2, x) && x == "peen");

assert(!that.get(-1, x) && x == "peen");

that.insert("ween");

assert(that.get(2, x) && x == "ween");

him.erase("4");

//aliasing with unite and subtract

Set kripp;

kripp.insert("1");

kripp.insert("6");

kripp.insert("4");

Set kapp;

subtract(him, kripp, kapp);

kapp.dump();

unite(him, kripp, kapp);

kapp.dump();

unite(him, kripp, him);

him.dump();

subtract(him, kripp, kripp);

kripp.dump();