In creating the Set class, I’ve made the Nodes work together as a non-circularly linked, doubly-linked list, with no dummy node. In managing the elements with insert() I’ve made it so that the value inserted into the linked list is sorted into the right spot from least to greatest (using the logic operations). Therefore, the head pointer at initialization points to null. Consequently, with more nodes, the starting node’s previous pointer is null, just as the last node’s next pointer is null. Refer to the image:

head

|

\_\_\_\_v\_\_\_ \_\_\_\_\_\_\_\_

| | | |

| “Hello” | | “World” |

|\_\_\_\_\_\_\_ | |\_\_\_\_\_\_\_ |

| ------------> | next = |

| | | nullptr |

|\_\_\_\_\_\_\_| |\_\_\_\_\_\_\_|

| prev = | | |

|\_nullptr\_| <-------------\_\_\_\_\_\_|

Pseudo code

~Set()

{

Loop:

Save next node

delete curr node

update next node using temp

}

operator=()

{

If the two sides of the = are different

Create a temp Set

Swap the temp

Return this

}

size()

{

Loop through set:

Increment count

Return count

}

insert()

{

If value is contained

Return false

If set is empty

add first element in Set

else If value comes before first element in Set

Add it to the front of Set

Else

Loop until at the last element:

If value comes before current element

Break

If is the last item in the Set

Add value to the end

Else

Add item to middle of Set

Return true

}

erase()

{

If value not contained in Set

Return false

If size is one

Delete the only present node

Return true

Loop until at the last element:

If val is equal to curr elements value

Break

If the node of interest is the first node

adjust the next/prev pointers at the front of Set

Else if it’s the last node

Adjust the next/prev pointers at the end of Set

Else

Adjust the pointers in the middle of the Set

Delete the node whose next/prev pointers were adjusted

Return true

}

contains()

{

bool Variable assumes the value is not in set

Loop through every element in Set:

If the curr val is value

Update var to true

Return bool var

}

get()

{

If i is out of bound of Set

Return false

Loop:

Until i th element in Set

Update value to the element

Return true

}

swap()

{

Temp pointer to head

Head is set to other’s head

Other’s head is set to temp pointer

}

unite()

{

If result is not empty

Loop through result:

Get the i th item in result

Erase it from result

Loop through s1:

Get the i th item in s1

Insert it into result

Loop through s2:

Get the i th item in s2

Insert it into result

}

subtract()

{

If result is not empty

Loop through result:

Get the i th item in result

Erase it from result

Loop through s1:

Get the i th item from s1

If s2 doesnt contain this item

Insert it into result

}

Test Cases

Set()

Set x;

Constructor is to be called

~Set()

When leaving scope of a if statement or loop, Set should be destroyed

Set(const Set &other);

Set x;

insert(1);

Set y = x;

Constructs a new Set identical to x

Set &operator=(const Set &rhs);

Set x;

insert(1);

Set y;

x = y;

Makes Set x identical to Set y.

int size() const;

Set x;

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

returns true for an empty set

insert(1);

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

returns the size for of the Set for a Set that has Items

bool empty() const;

Set x;

assert(x.empty());

returns true for an empty Set

insert(1);

assert(!x.empty());

returns false for a Set with items

bool insert(const ItemType &value);

Set x;

assert(x.insert(39));

returns true inserting into an empty Set

assert(x.insert(3));

returns true inserting at the top of the Set

Set y; y.insert(10);

assert(x.insert(12));

returns true inserting at the end of Set

assert(x.insert(11));

returns true inserting in the middle of Set

assert(!x.insert(10));

returns false inserting value that is already in Set

bool erase(const ItemType &value);

Set x;

assert(!x.erase(0));

returns false erasing a value from a empty Set

x.insert(0);

assert(!x.erase(12));

returns false erasing a value that doesn’t exists in Set

assert(x.erase(0)) ;

returns true erasing from a Set with one Node.

x.insert(1); x.insert(2);

assert(x.erase(2));

returns true erasing at the end of the list.

assert(x.erase(0));

returns true erasing at the top of the list.

assert(x.erase(1));

returns true erasing in the middle of the list.

bool contains(const ItemType &value) const

Set x;

assert(!x.contains(1);

Returns false when set is empty.

x.insert(0)

assert(!x.contains(1));

Returns false when set does not have value.

assert(x.contains(0));

Returns true when Set has one node whose val is value.

x.insert(1); x.insert(2);

assert(x.contains(0));

Returns true when Set’s value is found in the first Node.

assert(x.contains(1));

Returns true when Set’s value is found in the middle of Set.

assert(x.contains(2));

Returns true when Set’s value is found in the middle of Set.

bool get(int i, ItemType &value) const

Set a;

string value = "";

assert(!a.get(0, value));

returns false in empty Set, value is empty

a.insert("a");

assert(!a.get(1, value));

returns false, not found because i is greater than a's size

assert(!a.get(0, value));

returns true, value is now be "a"

assert(!a.get(-1, value));

returns false because i is negative

a.insert("b");

assert(a.get(1, value);

returns true, value is now"b"

void swap(Set &other)

Set b;

assert(a.swap(b));

returns true, when two empty sets

a.insert("a");

assert(a.swap(b));

returns true, b is has"a" and a has nothing

a.insert("b");

assert(a.swap(b));

returns true, b has "a" and "b", a has nothing

assert(a.swap(b));

returns true, a has "a" and "b", b has nothing

a.erase("b"); b.insert("b");

assert(a.swap(b));

returns true, b has"a", a has"b"

b.insert("c");

assert(a.swap(b));

returns true, b has"a" and "c", a has"b"

a.erase("b"); a.erase("a"); a.erase("c");

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

Set c;

assert(unite(a, b, c));

has three empty sets

c = a;

assert(unite(a, b, c));

three empty sets, result set is the same as a normal set

c = new Set(); a = b;

assert(unite(a, b, c));

three empty sets where s1 and s2 are the same

a.insert("a");

assert(unite(a, b, c));

c has "a"

a.erase("a"); b.insert("a");

assert(unite(a, b, c));

c has "a"

a.insert("a");

assert(unite(a, b, c));

c should has "a" only once

a.insert("b");

assert(unite(a, b, c));

c has "a" and "b"

a.erase("b"); b.insert("b");

assert(unite(a, b, c));

c should now contain "a" and "b"

a.erase("a"); b.erase("a"); b.erase("b");

void subtract(const Set &s1, const Set &s2, Set &result)

assert(subtract(a, b, c));

Three empty sets

c = a;

assert(subtract(a, b, c));

three empty sets where the result set is the same as a normal set

c = new Set();

a = b;

assert(subtract(a, b, c));

three empty sets where the s1 equals s2

a.insert("a");

assert(subtract(a, b, c));

c has"a"

b.insert("a");

assert(subtract(a, b, c));

c is still be empty

a.insert("b");

assert(subtract(a, b, c));

c has"b"

b.insert("c");

assert(subtract(a, b, c));

c has "b"