Christopher Kha

705101509

1/27/19

Project 2 report

1. **Design**

My doubly linked list consisted of a header pointer and several connected Nodes organized in ascending order. Each Node recorded a *value* of ItemType, a *next* pointer and a *prev* pointer. I ordered the list by placing them in the right location when insert() is called.

1. **Pseudocode**

**insert()**

If the value is already in the Set

Don’t do anything and return false

If the value belongs in the first position (if the first Node’s value is greater than value)

Create a new Node and add it to the front

Update next and prev pointers

Increment m\_size

Return true

Traverse through the list and get the position of the Node right before the Node whose value is greater than value

Create a new Node and add it to that position

Update next and prev pointers

Increment m\_size

Return true

**erase()**

If the value is in the first Node

Update pointers and delete the target Node

Decrement m\_size

Return true

Traverse through the list until we find the target Node or until the temporary pointer is a nullpointer

If the temporary pointer isn’t a nullpointer

Update pointers and delete the target Node

Decrement m\_size

Return true

Return false

**unite()**

If s2 and result aren’t aliases

Make result into a copy of s1

Traverse through s2, s2.size() times

Use get() to retrieve the value at the ith position

If result doesn’t currently contain that value, insert it

Else

Traverse through s1, s1.size() times

Use get() to retrieve the value at the ith position

If result doesn’t currently contain that value, insert it

**subtract()**

If s2 and result are aliases

Copy construct s2 and store it in temp

Make result into a copy of s1

Traverse through temp

If any value in temp is contained in result

Erase that value

Else

Make result into a copy of s1

Traverse through s2

If any value in temp is contained in result

Erase that value

**Copy constructor**

Set m\_size to 0

Set m\_head to nullptr

Traverse through src and use insert() to add the value of each Node to the new list

**Assignment operator**

If “this” and src reference the same object

Return \*this

Traverse through the current Set’s list and delete every Node

Set m\_size to 0

Set m\_head to nullptr

Traverse through src and use insert() to add the value of each Node to the new list

Return \*this

1. **Test cases**

using ItemType = std::string;

Set s; // Creating an empty set

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

assert(s.empty());

assert(s.insert("how")); // **insert** to empty Set

assert(s.insert("wow")); // insert to end

assert(s.insert("a2")); // insert to front

assert(s.insert("bow")); // insert to middle

assert(s.size() == 4); // **size()**

assert(!s.insert("a2") && s.size() == 4); // attempt to insert duplicate

assert(s.erase("a2") && s.size() == 3); // **remove** from front

assert(!s.erase("a2") && s.size() == 3); // nothing to remove

assert(s.erase("how") && s.size() == 2); // remove from middle

assert(s.erase("wow") && s.size() == 1); // remove from end

Set s2;

s2.insert("y"); // insert to front multiple times

s2.insert("m");

s2.insert("c");

s2.insert("a");

ItemType val = "";

// **get()**

assert(!s2.get(-1, val) && val == ""); // pos less than 0

assert(!s2.get(4, val) && val == ""); // pos equal to m\_size

assert(!s2.get(5, val) && val == ""); // pos greater than m\_size

assert(s2.get(0, val) && val == "a"); // getting the value at each position

assert(s2.get(1, val) && val == "c");

assert(s2.get(2, val) && val == "m");

assert(s2.get(3, val) && val == "y");

s.swap(s2); // **swap** two Sets

assert(s.contains("y") && s.contains("m") && s.contains("c") &&

s.contains("a") && s.size() == 4);

assert(s2.contains("bow") && s2.size() == 1);

Set s3; // swap an empty Set

s3.swap(s2);

assert(s2.empty());

assert(s3.contains("bow") && s3.size() == 1);

if(true){

Set s4 = s3; // **copy constructor**

s4.dump();

assert(s4.contains("bow") && s4.size() == 1);

Set s5 = s2; // copying an empty set

assert(s5.empty());

}

assert(s3.contains("bow") && s3.size() == 1); // **destructor**

// s3's linked list isn't deleted when s4 is destructed

Set s6;

Set s7;

s6.insert("yes");

s6.insert("or");

s6.insert("no");

s7.insert("why");

s7.insert("not");

// **assignment operator**

s7 = s6; // make a copy of s6

assert(s7.contains("yes") && s7.size() == 3);

Set s8;

Set s9;

Set s10;

s8.insert("a");

s8.insert("b");

s8.insert("c");

s9.insert("c");

s9.insert("d");

s9.insert("e");

// **unite()**

unite(s8, s9, s10); // s10 should contain "a", "b", "c", "d", "e"

assert(s10.contains("a") && s10.contains("b") && s10.contains("c")

&& s10.contains("d") && s10.contains("e") && s10.size() == 5);

unite(s8, s9, s8); // s1 and result are aliases

assert(s8.contains("a") && s8.contains("b") && s8.contains("c")

&& s8.contains("d") && s8.contains("e") && s8.size() == 5);

s8.insert("f");

unite(s8, s9, s9); // s2 and result are aliases

assert(s9.contains("a") && s9.contains("b") && s9.contains("c")

&& s9.contains("d") && s9.contains("e") && s9.contains("f") && s9.size() == 6);

Set s11;

Set s12;

unite(s11, s9, s12); // s1 is empty

assert(s12.contains("a") && s12.contains("b") && s12.contains("c")

&& s12.contains("d") && s12.contains("e") && s12.contains("f") && s12.size() == 6);

Set s13;

Set s14;

Set s15;

Set s16;

s13.insert("a");

s13.insert("b");

s13.insert("c");

s14.insert("b");

s14.insert("c");

s14.insert("d");

s14.insert("e");

subtract(s13, s14, s15); // **subtract()**

assert(s15.contains("a") && s15.size() == 1);

subtract(s16, s14, s15); // s1 is empty

assert(s15.empty());

subtract(s13, s16, s15); // s2 is empty

assert(s15.contains("a") && s15.contains("b") && s15.contains("c") && s15.size() == 3);

subtract(s13, s14, s13); // s1 and result are aliases

assert(s13.contains("a") && s13.size() == 1);

s13.insert("b");

s13.insert("c");

subtract(s13, s14, s14); // s2 and result are aliases

assert(s14.contains("a") && s14.size() == 1);

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

using ItemType = unsigned long;

Set s0; // **empty()**

assert(s0.empty());

assert(s0.size() == 0);

s0.insert(0); // **insert()**

s0.insert(1);

assert(!s0.insert(0) && s0.size() == 2); // inserting duplicates

assert(s0.erase(0) && s0.size() == 1); // **erase()**

assert(!s0.erase(0) && s0.size() == 1); // nothing to erase

s0.insert(2);

s0.insert(3);

s0.insert(4);

s0.insert(5);

s0.insert(6);

s0.insert(7);

assert(!s0.contains(0)); // **contains()**

assert(s0.contains(7));

assert(s0.contains(5));

assert(!s0.contains(8));

ItemType y = 10; // **get()**

assert(!s0.get(7, y) && y == 10); // not in range

assert(!s0.get(-1, y) && y == 10);

assert(!s0.get(8, y) && y == 10);

assert(s0.get(0, y) && y == 1); // first item

assert(s0.get(6, y) && y == 7); // last item