Sadhana Vadrevu

Nachenberg

Project 2 Report

**Set - Design**

Each Set has a size variable and a head and tail variable that point to the first and last item in the list, respectively. Each item is contained in a node, in a variable called value. The nodes also contain next and previous variables that point to the next node in the list and the previous node in the list, respectively. The last node in the Set’s next variable is set to nullptr. The same thing happens with the previous variable for the first node in the Set. The nodes are not in any particular order. If a Set is empty, the head and tail variables of that Set both point to the nullptr. If there is only one item in a particular Set, the head and tail pointers both point to the node containing that item.

**Psuedocode**

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

*if value is already in the set,*

*return false*

*allocate a new node*

*put value into new node*

*if the set is empty,*

*change “head” and “tail” to address of new node*

*set “next” and “previous” of new node to nullptr*

*otherwise,*

*link the current tail’s “next” value to address of new node*

*link the new node to the nullptr (“next”) and the current tail (“previous”)”*

*change “tail” to address of new node*

*increment size*

*return true*

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

*if the set is empty,*

*return false*

*if the item to be removed is the first in the set,*

*set “head” to the address of next node in the set*

*change the “previous” variable of the new head to nullptr*

*delete the node containing value*

*decrement size*

*return true*

*else if the item to be removed is the last in the set,*

*set “tail” to the address of the second-to-last node*

*change the “next” variable of the new tail to nullptr*

*delete the node containing value*

*decrement size*

*return true*

*otherwise,*

*traverse through the set until you reach the node just above the node containing value*

*if the node was found,*

*link the node above the node containing the value to the one below the node containing value and vice versa*

*delete the node containing value*

*decrement size*

*return true*

*otherwise, if the node was not found,*

*return false*

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

*if the set is empty,*

*return false*

*traverse through the each node in the set,*

*if the node contains the specified value,*

*return true*

*return false*

**Set::get(int pos, ItemType& value)**

*if pos is invalid,*

*return false*

*starting with the top node and for each node in the set,*

*check each node against every other node in the list,*

*each time the node is greater than the other nodes in the list,*

*increment a counter*

*if the counter is equal to pos,*

*break*

*the value is the value of the node the loop breaks on*

*return true*

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

*swap sizes*

*swap tail pointers*

*swap head pointers*

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

*create a new empty set*

*for every item in s1,*

*get each item and insert it into the new empty set*

*for every item in s2,*

*get each item and insert it into the new empty set*

*set result equal to the created empty set*

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

*create a new empty set*

*create a flag and set it to false*

*for every item in s1,*

*check against every item in s2,*

*if there is a match found*

*change flag to true*

*if flag if false (there was no match found)*

*insert item from s1 into empty set*

*set result equal to the created empty set*

**Test Cases**

Set ss; //default constructor

//for an empty set

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

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

assert(!ss.erase(“roti”)); //nothing to remove

assert(!ss.contains(“roti”)); //nothing in Set

ItemType x = “na”;

assert(!ss.get(0, x) && x == “na”); //test get when there’s nothing to get

assert(ss.insert("roti")); //test insert for an empty Set

assert(ss.insert("pita")); //test insert works for a non-empty Set

assert(!ss.insert(“roti”)); //check that insert won’t insert a word already in the Set

//for a non-empty Set

assert(ss.size() == 2); //test size

assert(ss.contains("pita")); //test contains when Set does contain word

assert(!ss.contains(“goldfish”)); //test contains when Set does not contain word

ItemType y = "laobing";

assert(!ss.get(4, y) && y == “laobing”); //test get when pos is out of range (y remains unchanged)

assert(ss.get(0, y) && y == "pita"); //test get

assert(ss.get(1, y) && y == "roti"); //test get

assert(ss.erase("roti")); //test erase

assert(ss.size() == 1); //test that erase properly decrements size

Set m;

assert(m.insert("ice cream"));

assert(m.insert("chocolate"));

assert(m.insert("pie"));

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

m.swap(ss); //swap m and ss

assert(ss.size() == 3 && m.size() == 1); //test that swap correctly switches the sizes of the Sets

assert(ss.get(2,x) && x == "pie"); //check that the swapped Set ss contains the items originally in m

Set newM = m; //copy constructor

//check that new object newM matches m

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

assert(newM.contains("pita"));

newM = ss; //assignment operator

//check that newM matches ss

assert(newM.size() == ss.size() && ss.size() == 3);

ItemType t = “hello”;

assert(newM.get(1, t) && t == “ice cream”);

ItemType k = “hello”;

assert(ss.get(1, k) && k == “ice cream”);

//test unite

Set result; //result is an empty Set

m.insert(“sourdough”);

ss.insert(“pita”); //now ss and m share an item

unite(ss, m, result);

assert(result.size() == 5); //check that result has the correct size

assert(result.get(0, k) && k == “chocolate”); //check that result has the correct items

unite(ss, ss, result); //check for proper functionality with aliasing in s1 and s2 and when result is a non-empty set

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

assert(result.contains(“pie”) && !result.contains(“roti”));

unite(m, ss, ss); //check for proper functionality with aliasing in s2 and result

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

assert(ss.get(5, k) && k == “sourdough”);

//test subtract

Set resultsub; //resultsub is an empty Set

Subtract(ss, m, resultsub);

assert(resultsub.size() == 4); //check that result has the correct size

assert(resultsub.contains(“pie”) && !resultsub.contains(“pita”)); //check that result has the correct items

subtract(ss, ss, resultsub); // check for proper functionality with aliasing in s1 and s2 and when result is a non-empty set

assert(resultsub.empty());

unite(ss, m, ss); //check for proper functionality with aliasing in s1 and result

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

assert(ss.contains(“pie”) && !resultsub.contains(“pita”));