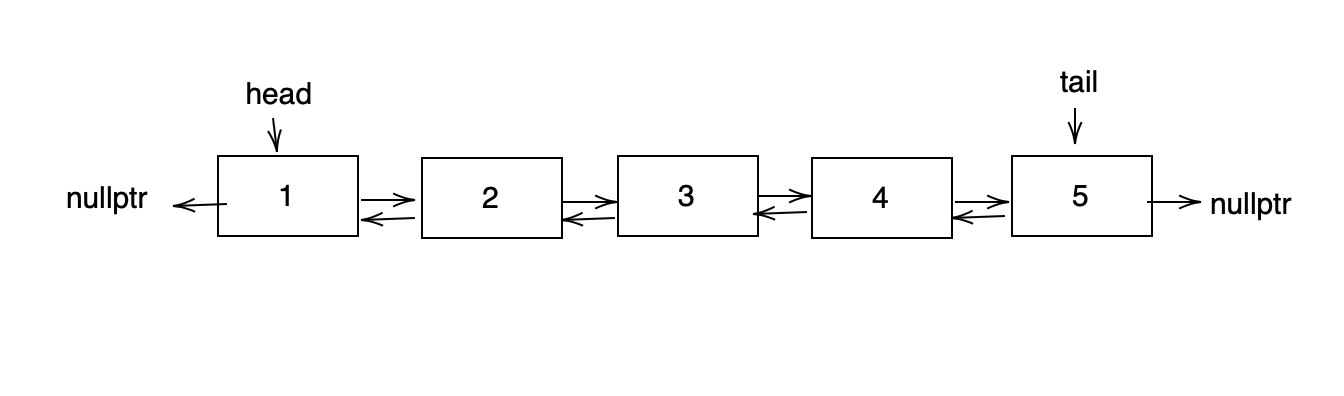
CS 32 Project 2 Report

Junhong Wang

# Design Description

The Set is implemented with doubly linked listed as shown below.

Set has 3 private member variables as the follow:

* m\_head: pointer to the head node
* m\_tail: pointer to the tail node
* m\_size: number of items/nodes in the Set

Node is structured as the follow:

* value: a place to store value
* next: pointer to next node
* prev: pointer to previous node

In order to make the get() function simple, items are inserted in such a way that they are in order. When there is only one item in the Set, head and tail points to the same node

# Pseudocode for Non-trivial Algorithms

|  |
| --- |
| **Set(const Set& other)** |
| Copy size  If other is empty, then set this be empty as well  while next node of other is not null:  Create a new node  Set the values of the new node  Add the new node to self |
| **~Set()** |
| If head is empty, then do nothing  While next node is not null:  Delete current node  Delete current node |
| **bool insert(const ItemType& value)** |
| If head is null, then insert at the head  If value is smaller than the value of head, then insert at the head  If value is greater than the value of tail, then insert at the tail  Else  while(true):  If the value of current node is same as value, then  return false  If the value is greater than current node and smaller than next node, then  Insert the new node  Break  Increment size  Return true |
| **bool erase(const ItemType& value)** |
| If the value of head is value, then erase head  If the value of tail is value, then erase tail  While current node is not null:  If the value of current node is greater than value, then return false  If the value of current node equals to value, then  Erase current node  Decrement size  Return true  Return false |
| **bool get(int pos, ItemType& value) const** |
| // The set is already sorted  If pos is invalid, then return false  Pick the node at position pos  Set the value be the value of the node picked  Return true |
| **Set& operator=(const Set& other)** |
| Copy the values from head until the smaller Set reaches the end  If self has same size as other, then done  If self is smaller, then  Create new nodes and assign values to each node  If self is bigger, then  Delete the redundant nodes |
| **unite(const Set& s1, const Set& s2, Set& result)** |
| Copy s1 to a placeholder  For each value in s2:  Insert value to placeholder  Set result be placeholder |
| **subtract(const Set& s1, const Set& s2, Set& result)** |
| Copy s1 to a placeholder  For each value in s2:  Erase value from placeholder  Set result be placeholder |

# Test Cases

The test code below are dependent, meaning to execute the code in a row, you must execute the code in previous code first. Sometimes you will see redundant tests. These are there to test other things in later rows.

|  |  |
| --- | --- |
| **Requirement Tested** | **Code** |
| Check constructor  Check size() with empty Set  Check empty() with empty Set  Check get() at index 0  Check value didn’t change due to get() | Set \*set = new Set();  std::string value = "default";  assert(set->size() == 0);  assert(set->empty() == true);  assert(set->get(0, value) == false);  assert(value == "default"); |
| Check insert into empty Set  Check insert in the middle  Check insert at the beginning  Check insert at the end  Check size() with non-empty Set  Check empty() with non-empty Set  Check the links of next and prev with dump() | set->insert("3");  set->insert("5");  set->insert("1");  set->insert("4");  set->insert("2");  set->insert("0");  set->insert("6");  assert(set->size() == 7);  assert(set->empty() == false);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "0 -> 1 -> 2 -> 3 -> 4 -> 5 -> 6 ->" << endl;  set->dump();  cout << "###" << endl; |
| Check get() works for any valid index  Check get() works for invalid index  (e.g -1, 7)  Check get() does not change value if the index is invalid | assert(set->get(-1, value) == false);  assert(value == "default");  assert(set->get(0, value) == true);  assert(value == "0");  assert(set->get(1, value) == true);  assert(value == "1");  assert(set->get(2, value) == true);  assert(value == "2");  assert(set->get(3, value) == true);  assert(value == "3");  assert(set->get(4, value) == true);  assert(value == "4");  assert(set->get(5, value) == true);  assert(value == "5");  assert(set->get(6, value) == true);  assert(value == "6");  assert(set->get(7, value) == false);  assert(value == "6"); |
| Check erase() for element in the middle of Set  Check size() is correctly updated when erasing an element in the middle of Set  Check erasing an element in the middle does not break the links of next and prev with dump() | assert(set->erase("3") == true);  assert(set->size() == 6);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "0 -> 1 -> 2 -> 4 -> 5 -> 6 ->" << endl;  set->dump();  cout << "###" << endl; |
| Check erase() for the first element of Set  Check size() is correctly updated when erasing the first element of Set  Check erasing the first element of Set does not break the links of next and prev with dump() | assert(set->erase("0") == true);  assert(set->size() == 5);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 2 -> 4 -> 5 -> 6 ->" << endl;  set->dump();  cout << "###" << endl; |
| Check erase() for the last element of Set  Check size() is correctly updated when erasing the last element of Set  Check erasing the last element of Set does not break the links of next and prev with dump() | assert(set->erase("6") == true);  assert(set->size() == 4);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 2 -> 4 -> 5 ->" << endl;  set->dump();  cout << "###" << endl; |
| Check erase() for an element that is larger than any element in the Set  Check size() does not change when erasing an element that is larger than any element in the Set (because it does not exist)  Check erasing an element that is larger than any element in the Set does not break the links of next and prev with dump() | assert(set->erase("7") == false);  assert(set->size() == 4);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 2 -> 4 -> 5 ->" << endl;  set->dump();  cout << "###" << endl; |
| Check erase() for an element that is larger and smaller than at least one element each in the Set, but is not in the Set  Check size() does not change when erasing an element described above.  Check erasing an element described above does not break the links of next and prev with dump() | assert(set->erase("3") == false);  assert(set->size() == 4);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 2 -> 4 -> 5 ->" << endl;  set->dump();  cout << "###" << endl; |
| Check erase() for an element that is smaller than any element in the Set.  Check size() does not change when erasing an element that is smaller than any element in the Set (because it does not exist)  Check erasing an element that is smaller than any element in the Set does not break the links of next and prev with dump() | assert(set->erase("0") == false);  assert(set->size() == 4);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 2 -> 4 -> 5 ->" << endl;  set->dump();  cout << "###" << endl; |
| Check erase() for an element in the middle of Set (This test is redundant, already checked before.) | assert(set->erase("2") == true);  assert(set->size() == 3);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 4 -> 5 ->" << endl;  set->dump();  cout << "###" << endl; |
| Check copy constructor  Check the size of the Set is indeed copied  Check the size of the original Set didn’t change.  Check if the contents of set2 are the same as those of set. | Set \*set2 = new Set(\*set);  assert(set->size() == 3);  assert(set2->size() == 3);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 4 -> 5 ->" << endl;  set->dump();  cout << "###" << endl;  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 4 -> 5 ->" << endl;  set2->dump();  cout << "###" << endl; |
| Check erasing an element in set2 does not affect set, but only affect set2. | assert(set2->erase("5") == true);  assert(set->size() == 3);  assert(set2->size() == 2);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 4 -> 5 ->" << endl;  set->dump();  cout << "###" << endl;  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 4 ->" << endl;  set2->dump();  cout << "###" << endl; |
| Check erase() works when erasing all the way to the end | assert(set2->erase("1") == true);  assert(set2->erase("4") == true);  assert(set2->size() == 0);  assert(set2->erase("0") == false);  assert(set2->size() == 0);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << endl;  set2->dump();  cout << "###" << endl; |
| Check operator overloading works when left hand side is empty  Check operator overloading doesn’t change set, but only changes set2. | \*set2 = \*set;  assert(set2->size() == 3);  assert(set->size() == 3);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 4 -> 5 ->" << endl;  set2->dump();  cout << "###" << endl;  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 4 -> 5 ->" << endl;  set->dump();  cout << "###" << endl; |
| Check erase() works (This test is redundant) | assert(set2->erase("4") == true);  assert(set2->erase("5") == true);  assert(set2->size() == 1);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 ->" << endl;  set2->dump();  cout << "###" << endl; |
| Check operator overloading works when right hand side is empty | Set \*set3 = new Set();  \*set2 = \*set3;  assert(set2->size() == 0);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << endl;  set2->dump();  cout << "###" << endl; |
| Check operator overloading works when right hand side is not empty  Check operator overloading works when left hand side is not empty | \*set2 = \*set;  assert(set2->size() == 3);  assert(set3->insert("100") == true);  assert(set3->size() == 1);  \*set2 = \*set3;  assert(set2->size() == 1);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "100 ->" << endl;  set2->dump();  cout << "###" << endl; |
| Check swap() works when the argument is empty Set | assert(set3->erase("100") == true);  assert(set3->size() == 0);  set2->swap(\*set3);  assert(set2->size() == 0);  assert(set3->size() == 1);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "100 ->" << endl;  set3->dump();  cout << "###" << endl;  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << endl;  set2->dump();  cout << "###" << endl; |
| Check swap() works when the caller is empty | set3->swap(\*set2);  assert(set2->size() == 1);  assert(set3->size() == 0);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "100 ->" << endl;  set2->dump();  cout << "###" << endl;  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << endl;  set3->dump();  cout << "###" << endl; |
| Check swap() works when both caller and argument are empty Set | Set \*set4 = new Set();  set4->swap(\*set3);  assert(set3->size() == 0);  assert(set4->size() == 0);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << endl;  set3->dump();  cout << "###" << endl;  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << endl;  set4->dump();  cout << "###" << endl; |
| Check swapping empty Set with itself works | set4->swap(\*set4);  assert(set4->size() == 0);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << endl;  set4->dump();  cout << "###" << endl; |
| Check swapping non-empty Set with itself works | set->swap(\*set);  assert(set->size() == 3);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "1 -> 4 -> 5 ->" << endl;  set->dump();  cout << "###" << endl; |
| Check insert() and size() work (This test is redundant) | Set \*s1 = new Set();  s1->insert("2");  s1->insert("8");  s1->insert("3");  s1->insert("9");  s1->insert("5");  assert(s1->size() == 5);  Set \*s2 = new Set();  s2->insert("6");  s2->insert("3");  s2->insert("8");  s2->insert("5");  s2->insert("9.9999");  assert(s2->size() == 5); |
| Check unite() works when s1 and s2 are not empty | unite(\*s1, \*s2, \*set);  assert(set->size() == 7);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "2 -> 3 -> 5 -> 6 -> 8 -> 9 -> 9.9999" << endl;  set->dump();  cout << "###" << endl; |
| Check unite() works when first argument and third argument are the same | unite(\*s1, \*s2, \*s1);  assert(s1->size() == 7);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "2 -> 3 -> 5 -> 6 -> 8 -> 9 -> 9.9999" << endl;  s1->dump();  cout << "###" << endl; |
| Check unite() works when second argument and third argument are the same. | unite(\*s2, \*s1, \*s1);  assert(s1->size() == 7);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "2 -> 3 -> 5 -> 6 -> 8 -> 9 -> 9.9999" << endl;  s1->dump();  cout << "###" << endl; |
| Check unite() works first and second arguments are the same | Set \*s3 = new Set();  unite(\*s1, \*s1, \*s3);  assert(s3->size() == 7);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "2 -> 3 -> 5 -> 6 -> 8 -> 9 -> 9.9999" << endl;  s3->dump();  cout << "###" << endl; |
| Check unite() works when all arguments are the same | unite(\*set, \*set, \*set);  assert(set->size() == 7);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "2 -> 3 -> 5 -> 6 -> 8 -> 9 -> 9.9999" << endl;  set->dump();  cout << "###" << endl; |
| Check unite() works when first argument is empty Set | unite(\*set3, \*set, \*set2);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "2 -> 3 -> 5 -> 6 -> 8 -> 9 -> 9.9999" << endl;  set2->dump();  cout << "###" << endl; |
| Check unite() works when second argument is empty Set | Set \*set5 = new Set();  unite(\*set2, \*set5, \*set2);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "2 -> 3 -> 5 -> 6 -> 8 -> 9 -> 9.9999" << endl;  set2->dump();  cout << "###" << endl; |
| Check subtract() works when s1 and s2 are not empty | subtract(\*s1, \*s2, \*set);  assert(set->size() == 2);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "2 -> 9 ->" << endl;  set->dump();  cout << "###" << endl; |
| Check subtract() works when second argument is empty Set | subtract(\*set, \*set3, \*set);  assert(set->size() == 2);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << "2 -> 9 ->" << endl;  set->dump();  cout << "###" << endl; |
| Check subtract() works when all arguments the same Set | subtract(\*set3, \*set3, \*set3);  assert(set->size() == 2);  cout << "###" << endl;  cout << "Expected printing from head" << endl;  cout << endl;  set3->dump();  cout << "###" << endl; |
| Check destructor  (Check memory leaks) | delete set;  delete set2;  delete set3;  delete set4;  delete set5;  delete s1;  delete s2;  delete s3; |