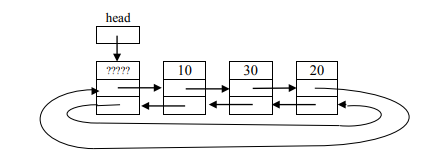
Kevin Wang

UID: 205209507

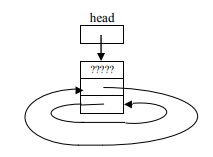
Project 2 Report

Description of Design

* I used a circular doubly linked list with a dummy head node. Each node object stores a value and 2 pointers, one that points to the node in front of it and one that points to the one before it. The last node in the set has a next pointer to the head dummy node and the head dummy node’s previous pointer points to the last item in the set.



* An empty set includes just the dummy head node that does not have a value. It’s next and previous node pointers are to itself.



* The order of the linked list is determined by the order in which the items are inserted. The insert function by default will add to the end of the list. There are no special cases to consider when inserting and erasing from the set.

Pseudocode for nontrivial algorithms

* Default constructor:
  + Initialize a set with a head dummy node that points to itself and has a size of 0
* Copy constructor
  + Initialize a new dummy head node
  + Create a pointer that refers to the current node
  + Create a loop with another pointer that traverses through the to be copied set and assigns values to a new node
  + Update the current node with values from the new node.
* Destructor
  + Create a loop that continues until there is only the dummy node in the set
    - Deallocate pointers to free up the node
    - Delete the node
  + Delete the dummy node
* Assignment operator
  + Perform an alias check
  + Copy-and-swap method
  + Return \*this
* Insert:
  + Return false if the set already contains the value
  + Initialize a pointer to the current node
  + Traverse to the end of the linked list
  + Allocate space for a new node and update its pointers
  + Increment the private data member for its size and return true
* Erase
  + Return false if the linked list already contains the value
  + Initialize a pointer to the current node and traverse to the location of where the value is found in the list
  + Deallocate its pointers and delete the node
  + Decrement the size and return true
* Contains
  + Traverse through the linked list and return true if the value is found
  + Return false otherwise
* Get:
  + Check that the supplied ‘pos’ is within bounds of our linked list, or else return false
  + Initialize a counter variable that keeps track of how many items in the list that a specific value is greater than
  + Create a loop that will continue until a match is found
    - Compare the values of one node with every other node in the list
    - Set value to the node that is greater than strictly pos number of items
    - Exit the loop and return true
* Swap:
  + Create temp variables for the size and head of the set
  + Swap each set’s size
  + Swap each set’s head pointer
* Unite:
  + Initialize an empty, temporary set
  + Using the get function, insert values from s1 and s2 into the temporary set
  + Assign the resulting set with values from tempSet
* Subtract:
  + Initialize an empty, temporary set
  + First use the unite function to add s1 and s2 into temporary set
  + Use the erase function to delete values from s2 in tempSet
  + Assign the resulting set with values from tempSet

Test cases:

// default constructor

Set ss;

// For an empty set:

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

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

assert(!ss.erase("a")); // nothing to remove

//testing insert/erase/size/and get functions

ss.insert("a");

assert(ss.size() == 1); //size increased by 1

assert(!ss.insert("a")); //cannot add the same item twice

ss.insert("b"); //add another item

assert(ss.size() == 2); //size correctly increased

ItemType x = "x";

assert(!ss.get(2, x) && x == "x"); //x is unchanged

assert(ss.get(0, x) && x == "a"); //x is changed to a

assert(!ss.erase("g")); //cannot erase a value that is not in the set

ss.erase("b");

assert(ss.size() == 1); //successfully erased a value from set

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

//check get function correctly returns correct value when the order in which the items are added is random

ss.insert("c");

ss.insert("b");

ss.insert("d");

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

ss.erase("d");

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

assert(ss.get(2, x) && x == "c");

testing the Swap function will ensure that our assignment operator and copy constructor are correct because of the copy-and-swap method

//initialize a new set

Set aa;

aa.insert("x");

aa.insert("y");

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

aa.dump();

//test swap function

ss.swap(aa);

assert(aa.size() == 3 && ss.size() == 2); //check sizes correctly swapped

cerr << "swapped values" << endl;

ss.dump();

aa.dump();

assert(ss.get(1, x) && x == "y"); //verify contents have been swapped

assert(aa.get(1, x) && x == "b");

Tests for unite and subtract functions

//test unite function

Set s1;

s1.insert("one");

s1.insert("two");

Set s2;

s2.insert("three");

s2.insert("four");

s2.insert("one"); //repeated values are not inserted

Set resultingSet; //if set is empty at first

unite(s1, s2, resultingSet);

resultingSet.dump();

unite(s1, s2, ss); //if resulting set had previous values

ss.dump();

//test subtract function

Set subtractSet; //when resulting set is empty

subtract(s1, s2, subtractSet);

subtractSet.dump();

subtract(s1, s2, ss); //when resulting set has previous values

ss.dump();