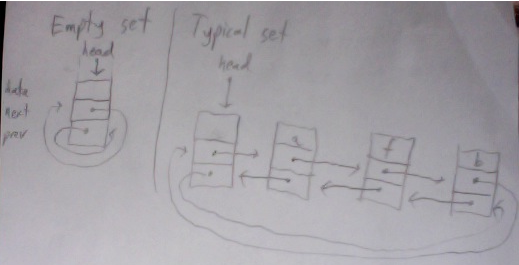
Kyle Kodani

Project 2

Report

For my implementation of the Set class, I used a doubly linked list. Every Set object has a head pointer that points to a dummy node. This node has no data associated with it. Each node (including the dummy node) has a pointer that points to the next node in the list, and a pointer that points to the previous node in the list. The pointers go all the way through the list and eventually point back to the dummy node, making the list circular. Every node besides the dummy node has some data associated with it. The size of the list is the number of nodes minus 1, because the dummy node is not included in the size. Example picture below:

**Pseudocode:**

Unite

*Create new default set and Itemtype*

*Insert elements of first set into new set*

*Insert elements of second set into new set*

*Set the result set to equal the new set*

Subtract

*Create new default set and Itemtype*

*Insert elements of first set into new set*

*Erase elements of second set from new set*

*Set the result set to equal the new set*

Get

Make sure i parameter is within the set’s size

*Using a temporary pointer, loop through the set until you point to the i position*

*Set the value parameter equal to the data at position i of the set*

Contains

*Using a temporary pointer, loop through the set*

*At each position, check to see if the value parameter is equal to the pointer’s data*

Erase

*Check to see if value parameter is not in set*

*Decrement size*

*Find where value is in the set*

*Adjust pointers to point around the node that contains value*

*Delete the node that contains value*

Insert

*Check to see if value is already in set*

*Create new node that stores value*

*Adjust new node’s pointers to point to head node and head’s previous node*

*Adjust pointers from head node and head’s previous node to point to new node*

*Increment size*

Assignment Operator

*Check to see if the two sets involved aren’t already equal*

*Create new set that is the same as the other set*

*Swap the new set with the this set*

Empty

*Check if size is 0*

Destructor

*Repeatedly, starting at head’s next node, until we have reached the head dummy node:*

*Advance one position through the set, then delete the previous node*

*Delete the head dummy node*

Copy Constructor

*Make head point to a new dummy node whose pointer’s point to itself*

*Insert data values from other set into this set*

Constructor

*Make head point to a new dummy node whose pointer’s point to itself*

Swap

*Switch the head pointers of the two sets*

*Switch the sizes of the two sets*

**Test Cases:**

These tests were performed when ItemType was an unsigned long

Set s1; // default constructor

//for an empty set

assert(s1.size() == 0); // test size

assert(s1.empty()); // test empty

assert(!s1.erase(1)); // nothing to remove, also tests contains

assert(s1.insert(1)); // test insert, also tests contains

// for a one item set

assert(s1.size() == 1); // test size

assert(!s1.empty()); // test empty

assert(!s1.erase(2)); // can't remove what isn't there, also tests contains

assert(s1.erase(1)); // successful erase, also tests contains

assert(s1.empty()); // set is now empty

assert(s1.insert(1)); // test insert

assert(s1.insert(2)); // test insert

// for a two item set

assert(!s1.insert(2)); // can't insert what is already there, also tests contains

// test get function

ItemType x=5;

assert(!s1.get(5, x)); // can't get something out of bounds

assert(x==5); // x unchanged by useless get call

assert(s1.get(0, x)); // test useful get call

assert(x==1 || x==2); // make sure get has changed x to something that is in the set

// testing swap

Set s2; // create a set to swap with

assert(s2.insert(3));

assert(s2.insert(4));

assert(s2.insert(5));

s1.swap(s2);

assert(s1.size()==3 && s2.size()==2); // successful size swap

assert(s1.erase(5)); // successful element swap

assert(s2.contains(2)); // successful element swap

s2.swap(s2); // swap with itself

assert(s2.size()==2); // no size change

assert(s1.get(1, x));

assert(x==3 || x==4); // successful element swap

// testing copy constructor

Set s3; // new set

Set s4(s3); // new set copied from s3

assert(s3.empty() && s4.empty()); // both sets should be empty

assert(s3.insert(1)); // fill s3 with data

assert(s3.insert(2));

assert(s3.insert(3));

Set s5(s3); // create set copied from s3

assert(s5.erase(3)); // successful element copy

assert(s3.contains(3)); // make sure s3 and s5 are distinct

// testing the assignment operator

// (this also tests the copy constructor and swap function)

Set s6; // create a new set

assert(s6.insert(1)); // fill s6 with data

assert(s6.insert(2));

assert(s6.insert(3));

Set s7; // create a new set

assert(s7.insert(4)); // fill s7 with data

s6=s7; // test assignment operator

assert(s7.contains(4) && s7.size()==1); // successful assignment

assert(s6.erase(4) && s7.contains(4)); // make sure s6 and s7 are distinct

s7=s7; // assignment to itself

assert(s7.contains(4) && s7.size()==1); // no change

// test unite

Set sa, sb, sc; // create new sets

assert(sa.insert(1)); // fill sets

assert(sb.insert(2));

assert(sc.insert(3));

unite(sa, sb, sc);

assert(sc.contains(1) && sc.contains(2)); // successful unite

assert(sc.size()==2); // successful unite

assert(!sc.contains(3)); // what sc had before is not still there

unite(sa, sb, sa); // aliasing test

assert(sa.contains(1) && sa.contains(2)); // successful unite

unite(sb, sb, sb); // aliasing test

assert(sc.contains(2)); // successful unite

// test subtract

Set sx, sy, sz; // create new sets

assert(sx.insert(1)); // fill sets

assert(sx.insert(2));

assert(sx.insert(3));

assert(sy.insert(2));

assert(sy.insert(3));

assert(sy.insert(4));

assert(sz.insert(5));

subtract(sx, sy, sz);

assert(sz.size()==1 && sz.contains(1)); // successful subtract

subtract(sx, sy, sx); // aliasing test

assert(sx.size()==1 && sx.contains(1)); // successful subtract

subtract(sy, sy, sy); // aliasing test

assert(sy.empty()); // successful subtract