My doubly-linked list is a simple design with a head node pointing to the first item of the list. Each item in the list has a value, a tally of how many times that that value occur, and pointers to both the item before and after it. Additionally, the class keeps a running total of how many items are in the linked list. It is not circular since the last item points to the nullptr as it’s next item.

Pseudocode:

bool Multiset::insert(const ItemType& value)

{

if list is empty

allocate a new node;

store value in new node;

set tally to 1;

increase total by 1;

link new node to the old top node;

link head pointer to new top node;

return true;

if list already contains value

use temp variable to repeatedly traverse the node that contains the value;

increase tally by 1;

increase total by 1;

return true;

else

use temp variable to repeatedly traverse to the current last node of the list;

allocate a new node;

store value in new node;

set tally to 1;

increase total by 1;

link current last node to the new node;

link new node to current last node;

link new node to nullptr;

return true;

}

int Multiset::erase(const ItemType& value)

{

if list is empty or does not contain value

return 0;

use a temp variable to traverse list in order to find the node that contains the value;

if the tally of that value is 1

decrease the total by 1;

if item is first in the list

connect head pointer to the second node in the list;

else

link the previous node to the next node;

if item is last in the list

connect second to last node in the list to nullptr;

else

link the next node to the previous node;

delete the pointer;

return 1;

if the tally of that value is more than 1

decrease the total by 1;

decrease the tally by 1;

return 1;

}

int Multiset::eraseAll(const ItemType& value)

{

create a variable to count number of items erased;

if list is empty or does not contain value

return 0;

use a temp variable to traverse list to find node that contains value;

set the total items erased to the tally of the node;

if node is first in list

set head pointer to second node in list;

else

set previous node to next node;

if node is last in list;

set previous node to nullptr;

else

set next node to previous node;

delete node;

decrease total by total items erased;

return total items erased;

}

void combine(const Multiset& ms1, const Multiset& ms2, Multiset& result)

{

create a temp Multiset;

repeatedly loop through each item of ms1;

use get function to retrieve value;

repeatedly loop “tally” times;

use insert value to put into temp;

repeatedly loop through each item of ms2;

use get function to retrieve value;

repeatedly loop “tally” times;

use insert value to put into temp;

set temp to result;

}

void subtract(const Multiset& ms1, const Multiset& ms2, Multiset& result)

{

create a temp Multiset;

repeatedly loop through each item of ms1;

use get function to retrieve value;

repeatedly loop “tally” times;

use insert value to put into temp;

repeatedly loop through each item of ms2;

use get function to retrieve value;

repeatedly loop “tally” times;

use erase value to remove values from temp;

set temp to result;

}

Test cases:

These tests were performed on a multiset of strings.

Multiset ms1, ms2; // default constructor

assert(ms1.size() == 0); // test size

assert(ms1.empty()); // test empty

assert(ms1.erase("nothing") == 0); // nothing to erase

assert(ms1.eraseAll("nothing") == 0); // nothing to eraseAll

assert(ms1.insert("alpha")); // test insert

assert(ms1.insert("beta"));

assert(ms1.insert("beta"));

assert(ms1.insert("gamma"));

assert(ms1.insert("delta"));

ms2 = ms1; // test assignment operator

assert(ms2.insert("epsilon"));

assert(ms2.size() == 6); // test size

assert(ms2.uniqueSize() == 5); // test uniqueSize

assert(!ms2.empty()); // test empty

assert(ms2.contains("alpha")); // test contains

assert(!ms2.contains("omega"));

assert(ms2.erase("beta") == 1); // test erase

assert(ms2.eraseAll("beta") == 1); // test eraseAll

assert(ms2.count("beta") == 0); // test count

Multiset ms3 = ms1; // test copy constructor

ms1.swap(ms3); // test swap

ItemType x;

ms1.get(0, x); // test get

assert(x == "alpha"); // test get changed x

combine(ms1, ms2, ms3); // test combine

// check if ms3 is the combination of ms1 and ms2

assert(ms3.contains("epsilon") && ms3.contains("beta"));

assert(ms3.count("alpha") == 2);

assert(ms3.size() == 9);

subtract(ms3, ms1, ms2);

// check if ms2 is the subtraction of ms3 - ms1

assert(!ms2.contains("beta") && ms2.contains("epsilon"));

assert(ms2.count("alpha") == 1);

assert(ms2.size() == 4);

// check for aliasing

subtract(ms1, ms1, ms1);

combine(ms2, ms2, ms2);

combine(ms1, ms2, ms1);

subtract(ms3, ms2, ms3);

subtract(ms3, ms3, ms2);

assert(ms2.empty());

cout << "Passed all tests" << endl;

These tests were performed on a multiset of unsigned longs.

Multiset ms1, ms2; // default constructor

assert(ms1.size() == 0); // test size

assert(ms1.empty()); // test empty

assert(ms1.erase(999) == 0); // nothing to erase

assert(ms1.eraseAll(999) == 0); // nothing to eraseAll

assert(ms1.insert(1)); // test insert

assert(ms1.insert(13));

assert(ms1.insert(13));

assert(ms1.insert(500));

assert(ms1.insert(789));

ms2 = ms1; // test assignment operator

assert(ms2.insert(88888));

assert(ms2.size() == 6); // test size

assert(ms2.uniqueSize() == 5); // test uniqueSize

assert(!ms2.empty()); // test empty

assert(ms2.contains(1)); // test contains

assert(!ms2.contains(0));

assert(ms2.erase(13) == 1); // test erase

assert(ms2.eraseAll(13) == 1); // test eraseAll

assert(ms2.count(13) == 0); // test count

Multiset ms3 = ms1; // test copy constructor

ms1.swap(ms3); // test swap

ItemType x;

ms1.get(0, x); // test get

assert(x == 1); // test get changed x

combine(ms1, ms2, ms3); // test combine

// check if ms3 is the combination of ms1 + ms2

assert(ms3.contains(88888) && ms3.contains(13));

assert(ms3.count(1) == 2);

assert(ms3.size() == 9);

subtract(ms3, ms1, ms2);

// check if ms2 is the subtraction of ms3 - ms1

assert(!ms2.contains(13) && ms2.contains(88888));

assert(ms2.count(1) == 1);

assert(ms2.size() == 4);

// check for aliasing

subtract(ms1, ms1, ms1);

combine(ms2, ms2, ms2);

combine(ms1, ms2, ms1);

subtract(ms3, ms2, ms3);

subtract(ms3, ms3, ms2);

assert(ms2.empty());

cout << "Passed all tests" << endl;