Design of Doubly Linked List:

My doubly linked list was not circular. The only way to access the first node in the linked list was through following the head pointer or through following the next nodes pointer to the previous node (in this case the first node). The struct for node was defined within the private section of class Multiset. Each node contains an ItemType variable, an integer variable named count, and a pointer to the previous node as well as the next node. The order of the list nodes is determined by the path of the pointers that link them.

Pseudocode for non-trivial functions:

Destructor:

initialize node pointer

loop through linked list

loop through linked list appropriately deleting nodes

delete final node

copy contructor:

loop through other’s nodes

insert other’s Itemtype variable into current linked list

insert other’s Itemtype variable into last node of current linked list

operator=:

guard against aliasing

make a temporary multiset

swap the current multiset with the temporary multiset

return \*this

insert:

if the linked list is empty

make a new node

intitalize head and tail pointer appropriately

initialize Itemtype variable to value

increment count appropriately

return true

otherwise loop through nodes

if variable equals value

increment count and return true

if value is not there you must make a new node

correctly adjust all pointers affected

Itemtype variable of new node gets the value variable that was passed in

Increment count of new node

erase:

loop through linked list

if Itemtype variable for particular node equal value

and the count is greater than one, decrement count

otherwise if count is 1

delete node but check the placement of the node first

for ex. If it’s the first node, a middle node, last node etc.

depending on the case adjust the pointers affected accordingly

return 1 if it deletes something, return 0 if it does not

eraseAll:

same as above function, but in this case you will delete the entire node in any case that a nodes Itemtype variable is equal to value. Adjust pointers appropriately depending on the position of the node

swap:

switch the head an tail pointers of this Multiset with the head and tail pointers of the other Multiset

combine:

check for aliasing

result gets m1

loop through m2 and add its elements to result

subtract:

check for aliasing

result gets m1

loop through m2 and call erase for each word(it will only erase if its present in result)

**Test cases**

Multiset ulms;

assert(ulms.insert(20));

assert(ulms.insert(10));

assert(ulms.insert(20));

assert(ulms.insert(30));

assert(ulms.insert(20));

assert(ulms.insert(10));

assert(ulms.size() == 6 && ulms.uniqueSize() == 3);

assert(ulms.count(10) == 2);

assert(ulms.count(20) == 3);

assert(ulms.count(30) == 1);

assert(ulms.count(40) == 0);

Multiset brady;

assert(brady.insert(40)); //test that insert works properly

assert(brady.insert(50));

brady.swap(ulms);

assert(!brady.contains(40)); //check that swap and contains work correctly

assert(!brady.contains(50));

assert(brady.contains(20));

assert(brady.size() == 6); //check that size and unique size

assert(brady.uniqueSize() == 3);

assert(ulms.contains(40) && ulms.contains(50));

assert(ulms.size() == ulms.uniqueSize());

assert(ulms.erase(50)); //test erase and size

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

assert(!ulms.contains(50) && ulms.contains(40));

assert(brady.eraseAll(20));

assert(brady.uniqueSize() == 2 && brady.size() == 3);

assert(brady.count(10) == 2); //test count

assert(brady.count(30) == 1);

Multiset Superbowl;

assert(Superbowl.insert(111));

assert(Superbowl.insert(111));

assert(Superbowl.insert(111));

assert(Superbowl.insert(222));

assert(Superbowl.insert(333));

Multiset AfcTitles;

assert(AfcTitles.insert(111));

assert(AfcTitles.insert(111));

assert(AfcTitles.insert(555));

assert(AfcTitles.insert(666));

assert(AfcTitles.insert(777));

Multiset result;

combine(Superbowl,AfcTitles, result); //test combine

ItemType temporary;

for(int i = 0; i < result.uniqueSize(); i++)

{

cout << result.get(i, temporary) << endl;

cout << temporary << endl;

}

ItemType holder; //see if multiset is still same after combine

for(int k = 0; k < Superbowl.uniqueSize(); k++)

{

cout << Superbowl.get(k, holder) << endl;

cout << holder << endl;

}

//Test subtract works correctly

Multiset subtractResult;

subtract(Superbowl, AfcTitles, subtractResult);

ItemType temporary2;

for(int i = 0; i < subtractResult.uniqueSize(); i++)

{

cout << subtractResult.get(i, temporary2) << endl;

cout << temporary2 << endl;

}

Multiset pets;

assert(pets.insert("dog"));

assert(pets.insert("cat"));

assert(pets.insert("cat"));

assert(pets.insert("gorilla"));

Multiset beasts;

assert(beasts.insert("wolf"));

assert(beasts.insert("dragon"));

assert(pets.erase("gorilla")); //to check if erase works when the count of the Itemtype in the node is 0

ItemType holder;

for(int i = 0; i <pets.uniqueSize(); i++)

{

cout << pets.get(i, holder) << endl;

cout << holder << endl;

}

assert(pets.eraseAll("cat")); //make sure erase all works properly

for(int i = 0; i <pets.uniqueSize(); i++)

{

cout << pets.get(i, holder) << endl;

cout << holder << endl;

}

(pets.swap(beasts)); //check if swap works correctly

for(int i = 0; i <pets.uniqueSize(); i++)

{

cout << pets.get(i, holder) << endl;

cout << holder << endl;

}

for(int i = 0; i <beasts.uniqueSize(); i++)

{

cout << beasts.get(i, holder) << endl;

cout << holder << endl;

}