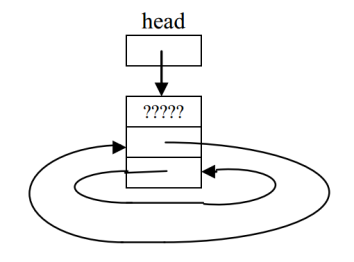
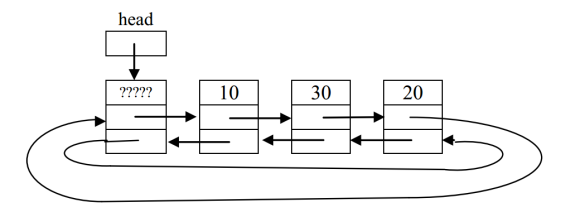
Project 2 Report

Patrick Tan

***Description***

My doubly-linked list had next and previous pointers for each node, along with a dummy node that pointed to m\_head and m\_tail. The list is circular, connected by the dummy node. The nodes contain m\_value, m\_count, and pointers to next and previous nodes. There is no particular order, other than new nodes being inserted will be added to the head of the doubly-linked list.

******

Typical Multiset Empty Multiset

***Psuedocode***

insert{

if(the value is already in multiset)

{

increment the count of the value

increment size

return true

}

otherwise:

{

Create a new Node

Set the value and the count to 1

Set next to the current m\_head

if: m\_head is NULL

{

Set the next value to the dummy Node

Set the next value for the dummy Node

}

if: m\_head is not NULL

Set the current m\_head's previous to the new Node

increment the size and the unique size

}

}

swap{

swap the dummy Node with the other dummy Node

swap m\_head with the other m\_head

swap m\_tail with the other m\_tail

swap the sizes

swap the unique sizes

}

~Multiset{

while(the next pointer is not the dummy)

delete the Node and traverse to the next Node

delete the dummy Node

}

Multiset(const Multiset& other){ //Copy Constructor

initialize the m\_head, dummy, and m\_tail

Set a node pointer to other m\_head

set a node pointer to m\_head

if: other.m\_head is not NULL

set the value and count of m\_head to other's m\_head

while: the next pointer of current is not the dummy

{

create a new node

set the value and count to the other's Node

insert the Node into the new multiset

set the next's and previous' to the correct Nodes

}

set the m\_tail and dummy to point to each other

}

operator=(const Multiset& rhs){ //assignment operator

if(rhs is not copying itself)

{

create a new multiset with rhs

swap the new multiset with the current multiset

}

}

find{

if: m\_head is NULL

return false

while: check does not equal dummy

{

if: check's value is the value we are searching for

return true

traverse to the next link

}

return false

}

doErase{

if: the value is present

{

if: we only remove one value and there is more than one value

decrement the count and the total size

find the total count of the value

subtract the total from size

decrement unique size by 1

remove the Node and set the corresponding pointers

delete the Node

}

}

combine{

set result to equal ms1

for: as many unique values in ms2

{

find the amount of a value and store the value

insert the value the amount of times it shows up

}

}

subtract{

set result to equal ms1

for: as many unique values in ms2

{

find the amount of a value and store the value

erase the value the amount of times it shows up

}

}

***Test Cases***

The tests were performed on a multiset of strings:

Multiset sb; // default constructor

ItemType test;

// For an empty multiset:

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

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

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

assert(sb.uniqueSize() == 0); // test uniqueSize

assert(sb.eraseAll("spice") == 0); // nothing to erase

assert(!sb.contains("anything")); // test contains

assert(sb.get(0, test) == 0); //test get, should not be able to find anything

//Tests basic functions

{

Multiset sms;

assert(sms.insert("cumin")); //insert into empty function

assert(sms.insert("turmeric")); //normal insertion

assert(sms.insert("cumin")); //insert a value already in

assert(sms.insert("coriander"));

assert(sms.insert("cumin"));

assert(sms.insert("turmeric"));

assert(sms.size() == 6 && sms.uniqueSize() == 3); //test size and uniqueSize of a filled multiset

assert(sms.count("turmeric") == 2); //tests count for multiple values

assert(sms.count("cumin") == 3); assert(sms.count("coriander") == 1); //tests count for single value

assert(sms.count("cardamom") == 0); //test count for a nonexistent value

assert(sms.get(2, test) == 1 || sms.get(2, test) == 2 || sms.get(2, test) == 3); //test get for tail value

assert(!sms.empty());

assert(sms.erase("cumin") == 1); //erase from head and single erase for multiple values

assert(sms.erase("cardamom") == 0); //erase from nonexistent value

assert(sms.erase("coriander") == 1); //erase and delete the tail

assert(sms.eraseAll("turmeric") == 2); //erase and delete with eraseAll

assert(sms.insert("turmeric"));

assert(sms.insert("turmeric"));

assert(sms.insert("coriander"));

assert(sms.insert("cumin"));

Multiset a(sms); //test copy constructor

assert(a.size() == 6 && a.uniqueSize() == 3); //test size and uniqueSize of a filled multiset

assert(a.count("turmeric") == 2); //tests count for multiple values

assert(a.count("cumin") == 3);

assert(a.count("coriander") == 1); //tests count for single value

assert(a.count("cardamom") == 0); //test count for a nonexistent value

assert(!a.empty());

assert(a.erase("cumin") == 1); //erase from head and single erase for multiple values

assert(a.erase("cardamom") == 0); //erase from nonexistent value

assert(a.erase("coriander") == 1); //erase and delete the tail

assert(a.eraseAll("turmeric") == 2); //erase and delete with eraseAll

} //test destructor

Multiset b;

Multiset c;

b.insert("a");

b.insert("b");

b.insert("c");

b.insert("d");

b.insert("b");

b.insert("b");

b.insert("d");

//test swap

b.swap(c);

assert(b.size() == 0);

assert(b.empty());

assert(c.count("b") == 3);

//test assignment operator

b = c;

assert(b.count("d") == 2);

assert(b.erase("c") == 1);

assert(c.eraseAll("b") == 3);

b.dump();

c.dump();

Multiset d(b); //test combine when b was set to d

combine(b,c,d); //test combine function

b.dump(); //values in b and values in c should add to values in d

c.dump();

d.dump();

Multiset e;

subtract(b,c,e); //test subtract function

b.dump(); //values in b subtracting values in c should result to values in e

c.dump();

e.dump();