## typical:

items

next

prev

items

next

prev

items

next

prev

head

items

next

prev

tail

Empty:

head nullptr

tail

The list is not circular and does not have a dummy node. The node is composed of three data members, an “items” struct which contains the name and number of the item, a next pointer pointing to the next node in the list and a prev pointer pointing to the previous node in the list.

It has a particular order. Every time there is a new item, a new node is added to the rear of the list.

constructor:

set initial number of kinds and number of items equal to 0;

set head and tail to nullptr;

destructor:

set p equal to head;

while there is still node in the list:

delete the current node;

jump to the next node

copy constructor:

set the number of kinds of items and total number of items equal to the number of kinds of items and total number of items of source;

if the source list is empty

set the current list empty;

while not every node from the source is copied

create a new node;

if current source node is head

set the new node to be head of the current list;

else

set the new node to be the next node;

set the new node’s previous node;

set the new node’s next node to be nullptr;

set the tail to be the new node;

set the new node’s value to be the source node’s value;

loop to the next node;

assignment operator:

if current list and source list are the same:

return current list;

set the number of kinds of items and total number of items equal to the number of kinds of items and total number of items of source;

delete nodes in the current list;

if the source list is empty

set the current list to be empty;

while not every node from the source is copied

create a new node;

if current source node is head

set the new node to be head of the current list;

else

set the new node to be the next node;

set the new node’s previous node;

set the new node’s next node to be nullptr;

set the tail to be the new node;

set the new node’s value to be the source node’s value;

loop to the next node;

return the current list;

Multiset::empty:

if number of kinds is 0, return true;

else return false;

Multiset::size:

return number of items;

Multiset::uniquesize;

return number of kinds;

Multiset::insert:

while p does not point to nullptr:

if there is any same kind of item in the list:

increase the number of that item by 1;

increase the total number of items by 1;

return true;

make p point to the next node;

create a new node;

if current list is empty:

make head and tail point to that new node;

else

set the originally tail node’s next to be the new node;

set the new node’s name;

set the new item’s number to be 1;

set the new node’s prev to be the originally tail node;

make the tail point to the new node;

increase the number of kinds and number of items by 1;

return true;

Multiset::erase;

while p does not point to nullptr:

if there is any same kind of the item in the list:

decrease the number of that item by 1;

decrease the total number of items by 1;

if the number of that item is equal to 0:

decrease the number of kinds by 1;

if there is only one node in list:

make head and tail point to nullptr;

else

if the current node is the tail:

make the prev node’s next point to nullptr;

make the tail point to the prev;

else

if current node is head

make head point to the next node;

make the next node’s prev point to nullptr;

else

make the prev node’s next point to the next node;

make the next node’s prev point to the prev node;

delete the current node;

return the number of items erased;

make p point to the next node;

return 0;

Multiset::eraseAll

while p does not point to nullptr:

if there is any same kind of the item in the list:

decrease number of items by the number of the current kind of item;

decrease number of kinds by 1;

if there is only one node in list:

make head and tail point to nullptr;

else

if the current node is the tail:

make the prev node’s next point to nullptr;

make the tail point to the prev;

else

if current node is head

make head point to the next node;

make the next node’s prev point to nullptr;

else

make the prev node’s next point to the next node;

make the next node’s prev point to the prev node;

delete the current node;

return number of items erased;

make p point to the next node;

return 0;

Multiset::contains:

while p does not point to nullptr:

if the node with the name found:

return true;

make p point to the next node;

return false;

Multiset::count

while p does not point to nullptr:

if the node with the name is found:

return the number of that item;

make p point to the next node;

return 0;

Multiset::get

if i is less than 0 or larger than or equal to the number of kinds:

return 0;

while p does not point to nullptr:

if it is not the ith item:

make p point to the next node;

else

break;

set value equal to the item’s name;

return the number of that item;

Multiset::swap

switch number of kinds, number of items, head and tail pointer of the two classes’ linked lists;

void combine:

make result equal to ms1;

loop over each item in ms2:

make n equal to the number of that item in ms2;

insert that item into result by n times;

void subtract:

make result equal to ms1;

loop over each item in ms2:

make n equal to the number of that item in ms2;

erase that item from result by n times;

## test cases:

Multiset ulms; // create a empty set with default constructor

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

assert(ulms.insert(20)); // insert item

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

assert(ulms.insert(10));

assert(ulms.insert(20));

assert(ulms.insert(30));

assert(ulms.insert(20));

assert(ulms.insert(10)); // insert more elements

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

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

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

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

assert(ulms.count(40) == 0); // check numbers of elements

Multiset b = ulms; // create another set using copy constructor

ItemType x;

assert(b.get(0, x) == 3 && x == 20); // test get

assert(b.get(1, x) == 2 && x == 10);

assert(b.get(0, x) == 3 && x == 20); // test if it generates the same result as the first get

assert(ulms.erase(20) == 1); // test erase

assert(ulms.erase(40) == 0); // nothing to erase

assert(ulms.size() == 5 && ulms.uniqueSize() == 3 && ulms.count(20) == 2); // test size after erasing

assert(ulms.eraseAll(20) == 2); // test eraseAll

assert(ulms.size() == 3 && ulms.uniqueSize() == 2 && ulms.count(20) == 0); // test size after erasing

assert(!ulms.contains(20) && ulms.contains(10) && ulms.contains(30)); // test contains

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

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

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

assert(b.count(40) == 0); // test if the copy constructor does the right thing, i.e., b is independent of ulms

b.swap(ulms);

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

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

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

assert(ulms.count(40) == 0); // test swap

Multiset c = b;

combine(ulms, b, c);

assert(c.count(10) == 4 && c.count(20) == 3 && c.count(30) == 2 && c.count(40) == 0); // test combine with c not empty

subtract(ulms, b, c);

assert(c.count(10) == 0 && c.count(20) == 3 && c.count(30) == 0 && c.count(40) == 0); // test subtract with c not empty

subtract(b, ulms, c);

assert(c.empty()); // test subtract with c finally being empty

b = ulms;

assert(b.eraseAll(20) == 3);

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

assert(b.count(20) == 0);

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

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

assert(ulms.count(20) == 3); // test assignment operator