Ashley Chang

504449890

CS32 Project 2 Report

1. My doubly linked list uses both a head and tail pointer, and inserts new unique items from the head/front. The list consists of nodes, each of which contain an item’s value, how many instances of the value exist (itemCount), and pointers to the next and previous nodes. The head’s previous pointer is linked to the null pointer, while the tail’s next pointer is linked to the null pointer as well. There is no order for the nodes, and traversing the list requires a temporary pointer; no default dummy nodes are used.

Example:

Individual = 6

uniqueCount = 3

100 200 300

NULL

Item: “B”

ItemCount: 3

Next:

Prev:

Item: “C”

ItemCount: 1

Next:

Prev:

NULL

Item: “A”

ItemCount: 2

Next:

Prev:

Tail:

300

Head: 100

An empty multiset:

Individual = 0

uniqueCount = 0

Head

NULL

Tail

2. Pseudocode

Constructor

//Set uniqueCount and individual to 0

//Set head and tail to nullptr

Destructor

//Create a pointer to traverse through array

//Delete item pointed to and move to next item

//Set head and tail to nullptr

Copy Constructor

//Use initializer list to set uniqueCount and individual to values from other

//For every unique element in the multiset,

//Create new creator node and set item/itemCount equal to

other value

//If the creator node is the first node being copied, set

head/next/previous to appropriate nodes/nullptr

//else link previous node’s next to creator and vice versa

//set creator’s next node to the nullptr

//set tail to last node

Assignment Operator

//As long as this object is not assigning itself,

//Delete memory currently allocated to multiset

//Assign member variables uniqueCount and individual from x

//For every unique element in the multiset,

//Create new creator node and set item/itemCount equal

to other value

//If creator node is first node set head/next/previous to appropriate nodes/nullptr

//Else link previous node’s next to creator and vice versa

//set creator’s next node to nullptr

//set tail to last node

//return reference to this

insert Function

//Use find function to search multiset and see if item already exists

//If yes, add to value count and increment individual count

//Return true

//If no, add new node p to front

//set p’s item/itemCount to appropriate values

//set p’s next item to head and previous to null ptr

//If list is empty and head doesn’t point to anything yet,

//set tail to p

//Else

//Set former head’s previous pointer to new node, then reset head

//Increment uniqueCount and individual

//return true

erase Function

//If list is empty, return 0

//Use find function to search multiset for value

//If found value,

//If there is only one instance of value, delete entire

node

//Set remove pointer/head/tail to appropriate node

//Link next/previous nodes to appropriate nodes

//Delete item pointed to by remove

//Decrement individual and unique Count and return 1

//More than one instance of value, so decrement value’s

itemCount and individual count

//return 1

//Else did not find value, so return 0

eraseAll Function

//If list is empty, return 0

//Use find function to search for value

//If found value, delete entire node

//Set remove pointer/head/tail to appropriate node

//Link next/previous nodes to appropriate nodes

//Record number of items to delete

//Adjust unique/individual count

//Delete remove pointer and return number deleted

//Didn’t find value, so return 0

get Function

//If list is empty or given integer is invalid, return 0

//Set getter (traversing) node to head

//While getter pointer is not nullptr or target node,

//Traverse list until target node

//Set value to item in getter node

//Return itemCount of target item

getMostFrequentValue Function

//If list is empty, return false

//Set default highest frequency to first item’s count

//Set getter (traversing) pointer to head

//Traverse through list

/If there is an item that has a higher frequency than current,

set that item to the new “most frequent value”

//Check if there is more than one value that has the highest frequency

//Traverse list again

//If another item has highest frequency, record

//If record counter is greater than 1, return false

//Else, only one value has highest frequency

//set value to most frequent item and return true

getLargestValue Function

//If list is empty, return false

//Set getter (traversing) node to head

//Create ItemType variable to store largest item (first item is default largest)

//Traverse list

//If find an item that is larger than current largest value, set

largest to that item

//Set value to largest

//Return true

getSecondLargestValue Function

//If list contains less than 2 unique items, return false

//Create Itemtype variable and set it to largest item

//Create Itemtype variable to store secondLargest item

//Set getter (traversing) pointer to head

//If head is the largest value, ignore it and start searching list from second item

//Traverse list

//If found true largest value, ignore it

//If found item that is larger than current (second) largest, set

secondLargest to that item

//Set value to secondLargest

//Return true

replace Function

//If list is empty, return false

//If new value is equal to original, return true;

//If original value does not exist in multiset, return false

//If new value is not already present in the multiset,

//replace all instances of original with new value

//Else if new value exists (and different from original),

//Add original items to new value count

//Delete extra node/items

//return true

countIf Function

//If operation sign is invalid, return -1

//Create count variable to record items that correspond to op

//Set getter (traversing) pointer to head

//For each operation sign,

//If op = ‘>’, record any items that

are larger than value in count, and return count

//If op = ‘=’, record any items that are equal to value in count, and return count

//If op = ‘<’, record any items that are less than value in count, and return count

swap Function

//Create temp pointer and set to head

//Swap this head/tail with the other head/tail

//Swap all private members (uniqueCount, individual) with the other private members using the swapAssist function

copyIntoOtherMultiset Function

//Set getter (traversing) pointer to head

//Traverse through array

//For each instance of original item, copy into other multiset

combine Function

//If ms1/ms2 is not equal to result,

//If result is not empty,

//Clear everything using eraseAll

//use copyIntoOtherMultiset to put ms1 & m2 in result

//else if &ms1 is the same as &result

//use copyIntoOtherMultiset to add ms2 items to ms1

//else if &ms2 is the same as &result

//use copyIntoOtherMultiset to add ms1 items to ms2

subtract Function

//If ms1 is not equal to result,

//If result is not empty,

//Clear everything using eraseAll

//For each unique item in ms1,

//Get value at index

//If ms2 contains matching items

//find difference and store in result

//Else if ms2 doesn’t have matching item

//Add whatever is in ms1 to result

//else if &ms1 is the same as &result

//Delete any matching ms2 items from ms1

}

3. TEST CASES

Multiset a; //Constructor

//Check that the multiset is really empty

assert(a.empty() && a.uniqueSize() == 0 && a.size() == 0); //Nothing inside

assert(a.erase("woo") == 0); //Nothing to erase

assert(!a.contains("moo")); //Doesn't contain anything

ItemType value = "";

assert(!a.getLargestValue(value) && value == ""); //No largest value to get

assert(!a.getSecondLargestValue(value) && value == ""); //No second largest value to get

assert(!a.getMostFrequentValue(value) && value == ""); //No frequent value to get

assert(a.get(0, value) == 0 && value == ""); //No item to get

//Insert items

assert(a.insert("moo") && a.insert("oink") && a.insert("quack")); //Check that all items were added successfully

assert(a.insert("oink") && a.insert("quack")); //Insert duplicates

//Check item counts

assert(a.contains("oink") && a.contains("moo") && a.contains("quack")); //Check that m contains items

assert(a.size() == 5 && a.uniqueSize() == 3); //Correct size and unique size

assert(a.count("moo") == 1 && a.count("oink") == 2 && a.count("quack") == 2); //Check instances of unique items

//Copy constructor

Multiset b = a;

//Multiset b tests (repeat of above)

assert(!b.empty()); //b is not empty (received copy from a)

assert(b.contains("moo") && b.contains("oink") && b.contains("quack")); //b contains items from a

assert(b.size() == 5 && a.uniqueSize() == 3); //Correct size and unique size

assert(b.count("moo") == 1 && b.count("oink") == 2 && b.count("quack") == 2); //Check instances of unique items

//Getting items

assert(!a.getMostFrequentValue(value) && value == ""); //Two values have the largest frequency, so fail

assert(a.getLargestValue(value) && value == "quack"); //Test get largest value

assert(a.getSecondLargestValue(value) && value == "oink"); //Test get second largest value

assert(b.getLargestValue(value) && value == "quack"); //Test again for b

assert(b.getSecondLargestValue(value) && value == "oink");

assert(a.insert("quack"));

assert(a.getMostFrequentValue(value) && value == "quack"); //Valid largest frequency

assert(a.get(0,value) && value == "quack"); //Get function successfully copies frequentValue into value

assert(a.get(1,value) && value == "oink");

assert(a.get(2,value) && value == "moo");

//Replace/swap items

assert(a.replace("quack", "meow")); //Replace an existing item

assert(a.contains("meow") && a.count("meow") == 3 && a.uniqueSize() == 3); //Verify replacement

assert(!a.replace("woof", "meow")); //Replace a non-existent item

assert(a.replace("meow", "oink")); //Replace existing item with another existing item

assert(a.count("oink") == 5 && a.count("moo") == 1 && a.uniqueSize() == 2); //Verify replacement

assert(b.insert("woof")); //Include new unique item in b

a.swap(b); //Perform swap

assert(a.contains("woof") && a.count("quack") == 2 && a.size() == 6); //Verify swap

assert(!b.contains("woof") && b.count("oink") == 5);

//Comparing values (countIf)

assert(a.countIf('=', "woof") == 1); //Equal to an existing item

assert(a.countIf('=', "squeak") == 0); //Equal to a non existent item

assert(a.countIf('>', "moo") == 5); //Greater than an existing item

assert(a.countIf('>', "buzz") == 6); //Greater than a non-existent item

assert(b.countIf('<', "moo") == 0); //Less than an existing item

assert(b.countIf('<', "oink") == 1);

assert(b.countIf('<', "zop") == 6); //Less than a non-existent item

//Erasing items

assert(b.erase("oink") == 1); //Erase single item

assert(b.count("oink") == 4); //Verify erasure

assert(b.erase("moo")); //Erase item with only one instance

assert(b.count("moo") == 0 && !b.contains("moo") && b.uniqueSize() == 1 && b.size() == 4); //Verify erasure

assert(b.eraseAll("oink") && b.empty()); //Erase all of item with multiple instances

assert(!b.eraseAll("moo")); //Erase non-existent item

//Assignment operator

b = a;

assert(b.contains("woof") && b.size() == 6); //Verify assignment

assert(b.count("moo") == 1);

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

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

//moo oink oink quack quack woof

//Non-member function testing

Multiset z; //New multiset to store combination

combine(a,b,z);

assert(z.size() == 12 && z.uniqueSize() == 4); //Verify combination

assert(z.count("woof") == 2 && z.count("moo") == 2 && z.count("oink") == 4 && z.count("quack") == 4); //Verify combination

Multiset q; //New multiset to store combination

q.insert("bloop"); //Insert items so multiset is not empty before using combine function

q.insert("neigh");

combine(a,b,q);

assert(!q.contains("bloop")); //Combine function removed previous items before combining

assert(!q.contains("neigh"));

assert(q.size() == 12); //Verify combination

assert(q.uniqueSize() == 4);

assert(q.count("woof") == 2 && q.count("moo") == 2 && q.count("oink") == 4 && q.count("quack") == 4);

Multiset r; //New multiset for subtraction

r.insert("one"); //Insert elements to be subtracted from

r.insert("two");

r.insert("three");

Multiset s; //New multiset for subtraction

s.insert("four"); //Insert elements to subtract with

s.insert("five");

s.insert("six");

Multiset t; //New multiset to store subtraction

subtract(r,s,t); //Subtraction--no same elements, so r remains the same

assert(t.uniqueSize() == 3 && t.size() == 3); //Verify subtraction

assert(t.count("one") == 1 && t.count("two") == 1 && t.count("three") == 1);

r.insert("four"); //Insert elements- multisets have an element in common

r.insert("four");

r.insert("four");

Multiset u; //New multiset to store subtraction

subtract(r,s,u); //Subtract s from r

assert(u.size() == 5 && u.count("four") == 2); //Verify subtraction--subtracted one "four" from r

s.insert("one"); //Insert elements- multisets have "one," "two," and "four" in common

s.insert("two");

subtract(r,s,u); //Subtract- Multiset not empty before subtract function

assert(u.count("three") == 1 && !u.contains("one") && !u.contains("two")); //Verify subtraction- one and two eliminated because only one each in r

assert(u.count("four") == 2); //Multiple "four"s, so one gets subtracted

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