Autessa Anoosheh

Professor Smallberg

CS 32

25 January 2022

Project 2 Report

**Description**

My doubly linked list is circular, with a dummy node at the head. When the list is empty there is a dummy node where the previous and next nodes point to the head (the dummy node itself). I have a variable m\_size to keep track of the amount of valid nodes in the list, and when the linked list is empty, only the dummy node, m\_size is 0.

**Pseudocode**

Trivial functions: empty, size, and insertOrUpdate (since it only calls insert and update functions)

//constructor

set size of list to zero

create dummy node for head, doubly linked and circular

//destructor

create temporary node equal to head

until temporary node next points to head

use another temporary node to delete each node

delete first temporary node

//copy constructor

create circular, doubly linked dummy node as head

check if size of old list is zero

if so, set new list size to zero

return

until temporary node equals the head

create new node

copy key and value from old list to new node

assign next and previous pointers to correct nodes

increment temporary pointers

link last node and head to make list circular

set size of new list to size of old list

//assignment operator

check if two maps are not already equal

create new map using copy constructor

use swap function

return this

//insert

check if list contains key

return false

otherwise

if size of list is zero

create new node

set its key and value to what was passed into function

set head pointers to new node

set new node pointers to head

increment size

return true

if size is greater than 0

set temp node to first valid node

until temp node equals head or temp node key is greater than key passed in

iterate through list

if temp node key is greater than key passed in

create new node

set its key and value to what was passed into function

set next and previous pointers to correct nodes

reset next pointer of previous node and previous pointer of next node

increment size

return true

if temp node equals head

create new node

set its key and value to what was passed into function

set previous pointer to correct node and next pointer to head

reset previous node’s next pointer and head’s previous pointer to new node

increment size

return true

//update

check if list contains key

set temp node to first valid node

until temp node key equals key passed in or temp node equals head

iterate through list

if temp node key equals key passed in and temp node does not equal head

set temp node’s value to value passed in

return true

return false

//erase

check if map contains key

set temp node to first valid node

until temp node key equals key passed in or temp node equals head

iterate through list

if temp node key equals key passed in and temp node does not equal head

reassign pointers of previous and next node to exclude temp node

delete temp node

decrement size

return true

return false

//contains

set temp node to first valid node

until temp node key equals key passed in or temp node equals head

iterate through list

if temp node key equals key passed in and temp node does not equal head

return true

if temp node equals head

return false

//get (2 parameters)

check if map contains key

until temp node key equals key passed in

iterate through list

if temp node key equals key passed in

set value parameter to value of temp node

return true

return false

//get (3 parameters)

if integer passed in is greater than or equal to size of list or less than zero

return false

set temp node equal to first valid node

repeatedly:

iterate through loop until reach the ith node

set key parameter to temp node key

set value parameter to temp node value

return true

//swap

swap sizes of maps using temp variable

swap head pointers of maps using temp pointer

//merge

set result map equal to m1

create bool flag

repeatedly:

get key and value for second map at jth node

if result map does not contain that key

insert key and value into result map

else

get value of that key from result map

if the values of the matching keys are not equal

erase node from result map

set flag to be true

return !flag

//reassign

set result map equal to m

get value of result map at first valid node and pass it to temp

repeatedly:

get value at next node

set value of previous node to value of next node

set key of previous node to key of next node

set value of last node to value of first node using temp

**Test Cases**

Tests performed on maps from strings to doubles

Map a; //create empty map

assert(a.empty()); //make sure map is empty

assert(!a.contains("")); //test that empty string is not in map, especially because dummy node may have a key value of empty string, ensures that contains function does not look at dummy node

assert(!a.erase("")); //test nothing to erase

assert(!a.update("", 21)); //nothing to update

ValueType v = 5;

assert(!a.get("", v) && v == 5); //test no value to get and that v stays unchanged

int i = 0;

KeyType k;

assert(!a.get(i, k, v) && v == 5); //test since index is equal to m\_size that no value exists and again v is unchanged

Map b;

b.insert("never", 27);

b.insert("gonna", 19);

b.insert("give", 23);

b.insert("you", 8);

b.insert("up", 42);

assert(b.size() == 5); //test to see if items inserted

assert(b.get(2, k, v) && k == "never" && v == 27); //test to see if correct value is retrieved, ensures items being inserted in ascending order

b.erase("never");

assert(b.size() == 4); //test to see if item is erased

assert(b.get(2, k, v) && k == "up" && v == 42); //test to see if retrieves correct item after never being erased

b.insertOrUpdate("gonna", 36); //test to see if it can handle updating

assert(b.get("gonna", v) && v == 36); //test to see if correctly update and value retrieved

assert(!b.contains("never")); //test contain function

assert(b.contains("you")); //test contain function

assert(!b.get(4, k, v)); //test that no item greater than 4 items in list since there are only 4 items in the list

assert(!b.empty()); //check to make sure not empty

b.swap(a);

assert(b.empty()); //make sure b is now empty

assert(!b.contains("you")); //make sure b does not have previous values, swap worked

assert(a.size() == 4); //check that a's size is 4

assert(a.get(2, k, v) && k == "up" && v == 42); //make sure that function that worked for b works for a

Map result;

result.insert("not", 0); //testing with a not empty result

assert(merge(b, a, result)); //test merge with m1 being empty map

result.print(); //print result to ensure correct merge

cout << endl; //for neatness

assert(merge(a, b, result)); //test merge with m2 being empty map

result.print(); //print result to ensure correct merge

cout << endl;

Map c; //new empty map

assert(merge(b, c, result)); //test merge with two empty maps

assert(result.empty()); //ensure result is empty

cout << endl;

Map d(a); //test copy constructor

assert(merge(d, a, result)); //test merge when m1 and m2 are equal

result.print(); //print result to ensure correct merge, essentially printing d

cout << endl;

d.print(); //ensure copy constructor worked

assert(d.size() == 4); //ensure size copied

Map e;

e.insert("let", 21);

e.insert("you", 8);

e.insert("down", 7);

cout << endl;

assert(merge(a, e, result)); //merge with same key with same value

result.print(); //ensure only one of the you key

cout << endl;

e.update("you", 9); //update value of you

assert(!merge(a, e, result)); //merge should be false because same key with different value

result.print(); //ensure you key is not in result

cout << endl;

Map r;

r.insert("not", 0);

result = r; //test assignment operator

assert(result.size() == 1); //make sure size was copied

result.print(); //make sure assignment operator works

cout << endl;

assert(merge(r, a, result)); //try merge where m1 equal result

result.print(); //ensure correct merge result

cout << endl;

reassign(a, result); //test reassign

result.print(); //make sure values get swapped

cout << endl;

e.update("you", 21);

reassign(e, result); //test reassign where values of different keys are the same

result.print(); //test correct output

cout << endl;

result = e;

reassign(e, result); //test reassign where m1 and result are the same

result.print(); //test for correct output

cout << endl;

reassign(b, result); //test reassign with empty m1

result.print(); //test for correct output, only head pointer