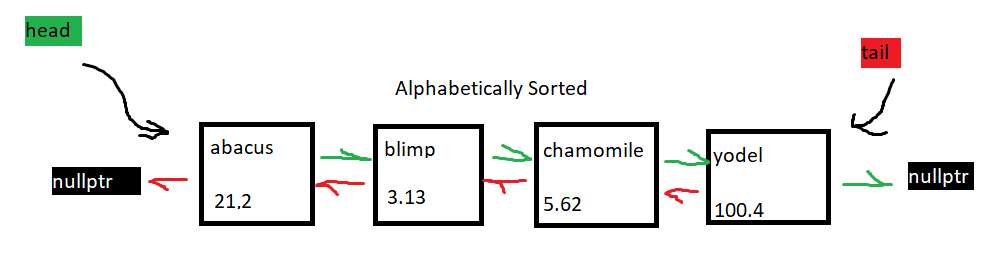
David Lin

1/25/2022

205770795

Project 2 – Double Exposure



* My doubly linked list implementation is alphabetically sorted to optimize performance of the get function. The list is not circular, nor does it have dummy nodes; however, it does have a tail in addition to its head. Each list node contains the key/value pair and pointers to the next and previous nodes.

**Pseudocode**

bool insert(const KeyType& key, const ValueType& value);

if map contains key,

can’t add duplicate key -> false

add item to alphabetically correct place and increment size

true

bool Map::update(const KeyType& key, const ValueType& value)

if map is empty or doesn’t contain key,

no keys to update -> false

repeatedly:

search for key

if key in map matches key,

replace old value with new value

true

key wasn’t found -> false

bool Map::insertOrUpdate(const KeyType& key, const ValueType& value)

if map contains key,

update value

true

insert the key/value pair

true

bool Map::erase(const KeyType& key)

if map is empty or doesn’t contain key,

no keys to erase -> false

repeatedly:

search for key

if key in map matches key,

break

if map only has one-item,

delete item

decrement size, head and tail are nullptr

true

if head node is being deleted

head now points to item after it

new head node has no previous node

delete head node and decrement size

true

if tail node is being deleted

tail now points to item before it

new tail node has no next node

delete tail node and decrement size

true

node in middle is being deleted

link adjacent nodes together

delete node and decrement size

true

bool Map::contains(const KeyType& key) const

false if empty

repeatedly:

search for key in map

if match is found

true

key wasn’t found->false

bool Map::get(const KeyType& key, ValueType& value) const

false if empty

repeatedly:

search for key in map

if match is found

copy value in map over to value

true

key wasn’t found->false

bool Map::get(int i, KeyType& key, ValueType& value) const

false if empty or invalid i value

find index of node that matches i

copy map’s key and value over

true

void Map::swap(Map& other)

swap size, tail, and head

addItem function produces an alphabetically sorted list

void Map::addItem(const KeyType& key, const ValueType& value)

if empty, add to front of list

if key is greater than first item’s key

add to front

if key is less than last item’s key

add to end

adding to the middle

repeatedly:

find node right before where to add

if the next key is greater than key

break

create new node with given key and value pair

adjust the prev and next pointers of current and adjacent nodes

bool merge(const Map& m1, const Map& m2, Map& result)

clear result

put all key value pairs of m1 in result

repeatedly:

insert key/value pairs of m2

if insertion fails,

if duplicate key/value pair matches existing pair in result,

do nothing

keys match, but values don’t

special key exists

return whether or not there’s a special key

void reassign(const Map& m, Map& result)

clear result

if less than 2 items in map, can’t reassign

if 2 items, swap values and put it in result

more than 2 items,

shift all values down by 1 key, insert in result

put first key’s value as the last key’s value, insert in result

**Test Cases**

// default constructor

Map m;

// For an empty map:

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

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

assert(!m.erase("Ricky")); // nothing to erase

assert(!m.update(“haha”, 20)); // nothing to update

assert(!m.contains(“Ricky”)); //empty map contains nothing

ValueType vorl = 34;

KeyType korl = “Ricky”;

assert(!m.get(korl, vorl)); //nothing to get

assert(!m.get(0,korl, vorl)); //nothing to get

//copy constructor

Map svap(m);

assert(svap.empty()); // copied empty map

svap.insert(“yea”,3);

svap.swap(m);

assert(svap.empty()); // filled map swaps with empty map and is now empty

Map ok;

ok.insert(“jump”,99);

assert(ok.contains(“jump”)); //test that map contains item

assert(ok.size() == 1); //test size on non-empty map

ok.erase(“jump”); //test erase

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

assert(!ok.contains(“jump”));

ok.insert(“jake”, 30);

ok.insert(“mba”, 42);

ok.insert(“msi”, 21);

ok.insert(“aba”,32);

assert(!ok.insert(“mba”,21)); // duplicate key, failed insertion

assert(ok.size()==4);

KeyType mh;

ValueType mal;

ok.get(“aba”, mal); // testing get

assert(mal == 32);

ok.get(1, mh, mal); //testing get(i)

assert(mh == “jake”);

ok.get(2, mh, mal);

assert(mh == “mba”);

ok.get(3, mh, mal);

assert(mh == “msi”);

assert(!ok.update(“dukck”,900)); // test update on absent key

assert(ok.insertOrUpdate(“dukck”,900)); //test insert or update

assert(ok.contains(“dukck”));

Map jam;

Map gem;

jam.insert(“joe”,2);

jam.insert(“kai”,0);

jam.insert(“mick”, 19);

jam.insert(“darn”,13);

gem.insert(“june”, 91);

gem.insert(“linoleum”, 54);

gem.swap(jam); //testing swap on non-empty arrays

assert(gem.contains(“joe”) && gem.contains(“kai”) && gem.contains(“mick”) && gem.contains(“darn”) && gem.size() == 4);

assert(jam.contains(“june”) && jam.contains(“linoleum”) && jam.size() == 2);

//testing assignment operator

Map mem;

Map mm;

mm.insert(“joe”,2);

mm.insert(“kai”,0);

mm.insert(“mick”, 19);

mm.insert(“darn”,13);

mem.insert(“june”, 91);

mem.insert(“linoleum”, 54);

mem = mm;

mm.erase(“mick”);

assert(mem.contains(“mick”)); //making sure that mem is a deep copy of mm

mm.update(“kai”,10);

KeyType ki;

ValueType val;

mem.get(“kai”, val);

assert(val == 0); //making sure that mem is a deep copy of mm

//testing copy constructor

Map mm;

mm.insert(“joe”,2);

mm.insert(“kai”,0);

mm.insert(“mick”, 19);

mm.insert(“darn”,13);

Map mem(mm);

mm.erase(“mick”);

assert(mem.contains(“mick”)); //making sure that mem is a deep copy of mm

mm.update(“kai”,10);

KeyType ki;

ValueType val;

mem.get(“kai”, val);

assert(val == 0); //making sure that mem is a deep copy of mm

Map olders; //testing that there is no special key in result

olders.insert("Fred", 123);

olders.insert("Ethel", 456);

olders.insert("Lucy", 654);

Map newers;

newers.insert("Lucy", 654);

newers.insert("Ricky", 321);

Map finale;

assert(merge(olders, newers, finale));

assert(finale.contains("Fred") && finale.contains("Ethel") && finale.contains("Ricky") && finale.contains("Lucy"));

Map olders; //testing that there is a special key in result

olders.insert("Fred", 123);

olders.insert("Ethel", 456);

olders.insert("Lucy", 654);

Map newers;

newers.insert("Lucy", 789);

newers.insert("Ricky", 321);

Map finale;

assert(!merge(olders, newers, finale));

assert(finale.contains("Fred") && finale.contains("Ethel") && finale.contains("Ricky") && !finale.contains("Lucy"));

//reassign testing

//one item

Map mak;

Map kil;

KeyType aoi;

ValueType ako;

mak.insert("a", 10);

reassign(mak,kil);

get(0, aoi, ako);

assert(ako == 10);

// no items & non-empty result map

Map mak;

Map kil;

kil.insert(“p”,4);

KeyType aoi;

ValueType ako;

reassign(mak,kil);

assert(kil.empty());

//multi item map

Map mak;

Map kil;

KeyType aoi;

ValueType ako;

mak.insert("a", 10);

mak.insert("b", 53);

mak.insert("c", 30);

mak.insert("d", 70);

reassign(mak, kil);

kil.get(0,aoi, ako);

assert(ako == 53);

kil.get(1,aoi, ako);

assert(ako == 30);

kil.get(2,aoi, ako);

assert(ako == 70);

kil.get(3,aoi, ako);

assert(ako == 10);