Project 2 Report – Ashvin Bondada

### Description of doubly-linked list:

The doubly linked list consists of “m\_Node”s which contain a Keytype key and ValueType value to hold the data, along with prev and next pointers per definition of doubly linked. The head’s prev ptr and tail’s next ptr both point to nullptr. I implemented a dummy head node to traverse the list. The list is ordered from the greatest ascii value in the head towards the lowest ascii value in the tail. Diagram below of typical and empty map:

Graphical user interface, application, Teams

Description automatically generated

# Pseudocode

## Map:erase:

if list empty

return false;

if head node must be deleted

if more than one node in list

adjust second node’s prev pointer to nullptr (it is the new head)

allocate killMe pointer to “node to be deleted”

delete node and subtract from total nodes

return true

traverse linked list

if found node above the key/value pair to delete or passed thru list

break

if did not pass thru last node (a key matching key was found)

allocate killMe pointer to “node to be deleted”

if not deleting the last node & keys match

adjust prev ptr of node below killMe to point to node above killMe

adjust next ptr of node above killMe to point to node/nullptr below killMe

delete killMe and subtract from total nodes

return true

return false if no matching key was found

## Reassign

create a temp map, copied from Map m

if map is not empty

From Map m list’s store tail’s key, head’s value in 1 temp variable each

repeatedly from bottom to top of Map m’s list

get a node’s value and node above’s key & insert new pair into temp map

get a node’s key/value pair and node above its’ key/value pair

insert the first node’s key and node above its’ value into temp map

insert tail node and head value (stored in temp variables) in to temp map

assign temp to result map

if map only has one pair or even empty

assign temp map to result

## Merge

define a new temp map

repeatedly from bottom to top of m2

get key/value pair from m2

get m1’s matching value for key in m2

if matching keys found but values don’t match

add to duplicate counter

skip adding key/value pair to result

insert key/value pair from m2 into temp map

repeatedly from bottom to top of m1

get key/value pair from m1

get m2’s matching value for key in m1

if matching keys found but values don’t match

add to duplicate counter

skip adding key/value pair to result

insert key/value pair from m1 into temp map

assign result to temp map

if duplicates found

return false

return true

## Map::Swap(Map& other)

switch head ptrs of both maps via temp head ptr var.

switch size of lists of both maps via temp var.

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

if i is a valid value

traverse list until “i” node found

copy values into parameter

return true

else

return false

## Map::insert(const KeyType& key, const ValueType& value)

If list is empty

add new node to front

set prev ptr to nullptr

add to node count

return true

If key already in list

return false

If key must be added to front

add to front

adjust prev/next ptrs;

add to node count

return true

If key must be added to end

add to end

adjust prev/next ptrs;

add to node count

return true

else (key must go in middle)

traverse list

if found node above where new node to be inserted

break

add to middle

adjust next/prev pointers of new node, the node above, and the node below

add to node count

return true

## Copy Constructor

repeatedly

get key/value pair from old map

insert pair into this map and add to node count

## Assignment Operator

If src map is not identical to map its being assigned to

Create a temp of the src and swap temp into map being assigned to

return this map

# Test Cases

The following test cases were tested when KeyType = std::string and ValueType = double. Note: linked list ordered from greater to least ascii values (Z’s closer to head, A’s closer to tail)

Map a, b;

### // test insert function

a.insert("Ashvin", 3.999); // tests adding to empty list

a.insert("Aaron", 4.890); // tests adding to front of list

a.insert("Zion", 3.556); // tests adding to end of list

a.insert("Bella", 3.987); // test adding to middle of list

assert(!a.insert("Ashvin", 3.098)); // fail, ashvin already exists

a.Dump();

// pass: a.Dump() prints in order of Zion, Bella, Ashvin, Aaron using next ptr and list in order of Aaron, Ashvin, Bella, Zion using prev ptr to verify doubly linked functionality

//test size and contains functionality

assert((a.size() == 4) && a.contains("Ashvin") && a.contains("Zion") && a.contains("Bella") && a.contains("Aaron"));

//should compile successfully showing size and contains work

### //test get(KeyType, ValueType)

ValueType u = 6;

assert(a.get("Zion", u) && u == 3.556); // pass: testing get head node, Zion is 3.556

assert(a.get("Aaron", u) && u == 4.890); // pass: testing get tail node, Aaron is 3.556

assert(a.get("Bella", u) && u == 3.987); // pass: testing get middle node, Bella is 3.556

### //test update function

ValueType v = 5;

ValueType w = 6;

a.update("Zion", v); // tests updating head of list - Zion currently 3.556

a.update("Aaron", v); // tests updating tail of list - Aaron currently 4.890

a.update("Bella", v); // tests updating middle of list - Bella currently 3.987

assert(!a.update("Aishu", 3.456)); // fail to update "Aishu" not in map

assert(a.get("Zion", w) && w == 5); // pass if function updated -> Zion is 5

assert(a.get("Aaron", w) && w == 5); // pass if function updated -> Aaron is 5

assert(a.get("Bella", w) && w == 5); // pass if function updated -> Bella is 5

### //test insertOrUpdate function

v = 10;

ValueType k;

a.insertOrUpdate("Landon", 4.000); // insert of update middle

a.insertOrUpdate("Ashvin", v); // update ashvin from 3.999 -> 10

a.insertOrUpdate("", 2.897); // inserts empty string, 2.987

a.insertOrUpdate("Johnny Appleseed", 2.900); // inserts new string

a.insertOrUpdate("Johnny Appleseed", 3.011); // Johnny Appleseed should be 3.011 now

assert(a.get("Johnny Appleseed", w) && w == 3.011); // pass if “update” and insert works

assert(a.get("Ashvin", k) && k == 10); // pass if just “update” works

assert(a.contains("Landon") && a.contains("")); // pass if update works

assert(a.size() == 7); // tests running total of nodes

### //test erase function

a.insert("Aaron", 4.989);

a.insert("Zzedd", 4.000);

assert(!a.erase("zzedd")); // fail -> case sensitive

assert(a.erase("")); // tests erase works for empty string

a.erase("Aaron"); // test erase for tail node

a.erase("Johnny Appleseed"); // test erase for middle node

a.erase("Zzedd"); // test erase for head node

assert(a.size() == 4 && (!a.contains("")) && (!a.contains("Aaron")) && (!a.contains("Johnny Appleseed"))); // pass if delete works

### //test 2nd get function

a.insert("", 2.897); // test if insert takes empty string and sorts it to tail

KeyType j;

a.get(0, j, k); // tests “get” for tail node of linked list

assert(j == "" && k == 2.897); // pass

a.get(2, j, k); // tests “get” for middle node of linked list

assert(j == "Bella" && k == 5); // pass

a.get(4, j, k); // tests “get” for head node of linked list

assert(j == "Zion" && k == 5); // pass

assert(!a.get(5, j, k)); // should fail because there are only 5 nodes in the map

### //test swap function

b.insert("Amanda", 2.895);

b.insert("John", 3.348);

b.insert("Abu", 2.136);

assert(a.size() == 5 && a.contains("") && a.contains("Ashvin") && a.contains("Bella") && a.contains("Landon") && a.contains("Zion"));

assert(b.size() == 3 && b.contains("Amanda") && b.contains("John") && b.contains("Abu"));

a.swap(b); // swaps a and b

assert(b.size() == 5 && b.contains("") && b.contains("Ashvin") && b.contains("Bella") && b.contains("Landon") && b.contains("Zion"));

assert(a.size() == 3 && a.contains("Amanda") && a.contains("John") && a.contains("Abu"));

b.swap(a); // should swap back into original form, assert below should pass if true

assert(a.size() == 5 && a.contains("") && a.contains("Ashvin") && a.contains("Bella") && a.contains("Landon") && a.contains("Zion"));

assert(b.size() == 3 && b.contains("Amanda") && b.contains("John") && b.contains("Abu"));

### // test copy constructor

Map c(b);

b.erase("Amanda");

// assert tests if b does not contain Amanda but c does contain Amanda and size func

assert(b.size() == 2 && (!b.contains("Amanda")) && b.contains("John") && b.contains("Abu"));

assert(c.size() == 3 && c.contains("Amanda") && c.contains("John") && c.contains("Abu"));

Map e; // testing empty map

Map f(e);

assert((e.size() == 0) && f.size() == 0); // both sizes should be 0

### // test assignment operator

assert(c.size() == 3 && c.contains("Amanda") && c.contains("John") && c.contains("Abu")); // verify c not equal to a

c = a; // test for 2 typical maps

// test a and c with the same arguments -> both should pass if c = a

assert(a.size() == 5 && a.contains("") && a.contains("Ashvin") && a.contains("Bella") && a.contains("Landon") && a.contains("Zion"));

assert(c.size() == 5 && c.contains("") && c.contains("Ashvin") && c.contains("Bella") && c.contains("Landon") && c.contains("Zion"));

Map g; // trying empty map

assert(g.size() == 0); // verify g is empty

c = g;

assert((c.size() == 0) && g.size() == 0); // test passes if g is copied over

### //test merge function

Map o, p, finalmap;

o.insert("Widow", 1); // tests head node in "o" with EQUAL key in "p", UNEQUAL value

o.insert("Banner", 1); // tests tail node in "o" with EQUAL key in "p", UNEQUAL value

o.insert("Purple", 1); // tests middle node in "o" with EQUAL key in "p", UNEQUAL value

o.insert("Caillou", 1);

o.insert("Neo", 3.987); // tests node in 'o' with EQUAL key & same EQUAL in "p"

o.insert("Youtube", 1);

o.insert("Friends", 1);

o.insert("Power", 1);

p.insert("Widow", 0); // equal in key "o"

p.insert("Banner", 0); // equal in key "o"

p.insert("Purple", 0); // equal in key "o"

p.insert("Chris", 0);

p.insert("Neo", 3.987); // equal in key and pair in "o"

p.insert("Tony", 0);

finalmap.insert("Bruce", 2); // final map is not empty for this test

finalmap.insert("Natasha", 2);

finalmap.insert("Pratt", 2);

finalmap.insert("ashvin", 2);

merge(o, p, finalmap); // tests merge if 1)same key/pair value in m1 & m2, 2)same key &

// diff value pair, 3)result is not empty

ValueType neo;

// finalmap should not have Widow, Purple, Banner, but have Neo, 3.987 pair, and should

// NOT have finalmap's original pairs too

assert(finalmap.contains("Youtube") && finalmap.contains("Tony") && finalmap.contains("Power") && finalmap.get("Neo", neo) && neo == 3.987 && finalmap.contains("Friends") && finalmap.contains("Chris") && finalmap.contains("Caillou") && !finalmap.contains("Widow") && !finalmap.contains("Purple") && !finalmap.contains("Banner") );

Map r; // test merge if result map is empty

merge(o, p, r);

assert(r.contains("Youtube") && r.contains("Tony") && r.contains("Power") && r.get("Neo", neo) && neo == 3.987 && r.contains("Friends") && r.contains("Chris") && r.contains("Caillou") && !r.contains("Widow") && !r.contains("Purple") && !r.contains("Banner") );

Map h; // test merge if m1 = m2 = result = Map h

h.insert("Bruce", 2);

h.insert("Natasha", 2);

h.insert("Pratt", 2);

// verify h is same as h after reassign

assert(h.contains("Bruce") && h.contains("Natasha") && h.contains("Pratt"));

merge(h, h, h); // test all parameters point to same func

assert(h.contains("Bruce") && h.contains("Natasha") && h.contains("Pratt"));

### //test reassign function

Map t;

t.insert("Banner", 0);

t.insert("Power", 4);

t.insert("Caillou", 1);

t.insert("Friends", 2);

t.insert("Neo", 3);

ValueType z;

assert(t.get("Power", z) && z == 4); // values are in order from 4 -> 0

assert(t.get("Neo", z) && z == 3);

assert(t.get("Friends", z) && z == 2);

assert(t.get("Caillou", z) && z == 1);

assert(t.get("Banner", z) && z == 0);

reassign(t, t); // reassign function programmed to swap every value by shifting up

assert(t.get("Power", z) && z == 3); // values have now shifted up 1

assert(t.get("Neo", z) && z == 2);

assert(t.get("Friends", z) && z == 1);

assert(t.get("Caillou", z) && z == 0);

assert(t.get("Banner", z) && z == 4);

reassign(t, t); // test reassign again

assert(t.get("Power", z) && z == 2); // values shifted up 1 again

assert(t.get("Neo", z) && z == 1);

assert(t.get("Friends", z) && z == 0);

assert(t.get("Caillou", z) && z == 4);

assert(t.get("Banner", z) && z == 3);

Map y; // test reassign for just one pair

y.insert("Bose", 3);

reassign(y, y);

assert(y.contains("Bose"));

Map q; // checking for empty map

reassign(q, q);

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