Vanessa Lee

Lecture 1

504766034

Project 2 Report

The doubly-linked list is not circular. Instead, it has a head and tail pointer pointing to the start and the end of the list respectively. It does not use a dummy node. When a list is empty, both the head and tail pointers point to a nullptr. The list is not in any particular order.

NULLPTR

HEAD

TAIL

VALUE

NEXT

PREV

VALUE

NEXT

PREV

VALUE

NEXT

PREV

VALUE

NEXT

PREV

TAIL

HEAD

Pseudocode:

Map::Map() {

Set m\_size to 0;

Set head and tail pointers to null;

}

Map::~Map() {

Traverse through linked list

Delete node;

}

Map::Map(const Map& m) {

Set m\_size = 0;

Set head and tail pointers to null;

Traverse through Map m {

Insert key and value pairs from Map m;

}

}

Map& Map::operator=(const Map &src) {

If this and src are the same

Return this;

Create temp map for src;

Swap temp map with this map;

Return this;

}

bool Map::insert(const std::string &key, const double &value) {

If key exists in map

Return false;

Create new node;

Set key and value parameters into new node;

If list is empty {

Head and tail points to new node;

Set next and prev pointers to null;

}

Else {

Insert node at front of list;

}

Increment size;

Return true;

}

bool Map::update(const std::string &key, const double &value) {

If key exists in map

Return false;

Repeatedly:

If p's next points to node containing key;

Break;

Point to next node;

Set node's value to new value;

}

bool Map::insertOrUpdate(const std::string &key, const double &value) {

Insert key if not already in map;

Otherwise update key and value;

Return true;

}

bool Map::erase(const std::string &key) {

If key does not exist in map

Return false;

Find node containing key;

Delete targeted node;

Reassign pointers;

Decrease size;

Return true;

}

bool Map::contains (const std::string &key) {

Traverse through list {

If key matches, return true;

Go to next node;

}

Otherwise return false;

}

bool Map::get(const std::string &key, double &value) {

If key does not exist in map

Return false;

Traverse through list {

If key matches, break;

Go to next node;

}

Set node value to value;

Return true;

}

bool Map::get(int i, std::string &key, double &value) {

If key does not exist in map

Return false;

Find the i-th node;

Set key and value to key and value parameters;

Return true;

}

void Map::swap(Map &other) {

Swap head pointers;

Swap tail pointers;

Swap capacities;

}

bool combine (const Map &m1, const Map &m2, Map &result) {

While (result is not empty)

Erase all the nodes;

For each node in map 1 {

Get key data and check if map 2 contains same key;

If map 2 does not contain same key

Insert key and value

Else check the values

If the values are the same, insert into result

Else false;

}

For each node in map 2 {

Get key data and check if map 1 contains same key;

If map 1 does not contain key

Insert the key and value into result;

}

}

void subtract (const Map &m1, const Map &m2, Map &result) {

While (result is not empty)

Erase all the nodes;

For each node in map 1 {

If node is in not in map 2

Insert into result;

}

}

Test Cases:

Map m1; // calls default constructor

assert(m1.empty() && m1.size() == 0); // default constructor creates empty map

assert(!m1.erase("Nothing to erase")); // can't erase an empty map

assert(m1.insert("G-Dragon", 88));

assert(m1.insert("Taeyang", 88));

assert(!m1.insert("G-Dragon", 1988)); // checks against adding two of the same key

assert(!m1.empty()); // map is no longer empty

assert(m1.contains("G-Dragon") && m1.contains("Taeyang")); // checks if keys are inserted properly

assert(m1.size() == 2); // insert function properly increases size

ValueType v;

m1.get("G-Dragon", v);

assert(v == 88); // checks that value was not changed when we tried to insert the same key twice

Map m2 = m1; // calls copy contructor;

assert(m2.contains("G-Dragon"));

assert(m2.size() == 2);

// checks that map was properly copied

Map m3;

KeyType bangtan[7] = {"Jungkook", "Taehyung", "Yoongi", "Jimin", "Hoseok", "Namjoon", "Seokjin"};

for (int i = 0; i < 7; i++)

assert(m3.insert(bangtan[i], i+10));

ValueType v2;

m3.get("Taehyung", v2);

assert (v2 == 11); // checks inserted value

assert(m3.erase("Seokjin") && m3.size() == 6); // checks that last node can be erased

assert(!m3.erase("Seokjin")); // can't erase a node that doesn't exist

assert(m3.erase("Jungkook")); // checks that first node can be erased

assert(m3.erase("Jimin")); // checks that middle node cna be erased

assert(m3.insertOrUpdate("Taehyung", 95)); // updates current key in map

assert(!m3.update("G-Dragon", 88)); // key does not exist in map

assert(m3.insertOrUpdate("BigHit", 2013) && m3.contains("BigHit") && m3.size() == 5);

// map does not contain value, so it should insert into map

m3.get("Taehyung", v2);

assert(v2 == 95); // value should have been changed with update function

m3 = m1; // calls assignment operator

assert(m3.contains("G-Dragon") && m3.size() == 2);

assert(!m3.insert("Taeyang", 1988)); // key already exists

assert(m1.contains("G-Dragon") && m1.size() == 2);

// checks that map was properly assigned

KeyType bb[5] = {"Daesung", "Seungri", "G-Dragon", "TOP", "Taeyang"};

Map b;

for (int i = 0; i < 5; i++)

assert(b.insert(bb[i], i));

Map bb2;

KeyType duo[3] = {"G-Dragon", "TOP", "Taeyang"};

for (int i = 0; i < 3; i++)

assert(bb2.insert(duo[i], i));

Map result;

assert(result.empty()); // passing in empty result

assert(!combine(b, bb2, result));

// values of the keys are different, so returns false

KeyType maknaes[2] = {"Daesung", "Seungri"};

Map bb3;

for (int i = 0; i < 2; i++)

assert(bb3.insert(maknaes[i], i));

assert(combine(b, bb3, result));

assert(!result.empty()); // result not empty from before

assert(result.contains("G-Dragon") && result.contains("Daesung"));

assert(result.size() == 5);

// checks that combine function works properly

subtract(b, bb2, result);

// result is not empty anymore, must not contain previous result

assert(!result.contains("G-Dragon"));

assert(!result.contains("TOP"));

assert(result.contains("Seungri") && result.contains("Daesung"));

assert(result.size() == 2);

// checks that subtract function works properly

bb2.swap(bb3);

assert(bb2.contains("Daesung") && bb2.contains("Seungri"));

assert(bb3.contains("G-Dragon") && bb3.contains("TOP") && bb3.contains("Taeyang"));

assert(bb2.size() == 2 && bb3.size() == 3);

// checks swap function