Lawrence Ouyang 504128219 CS32 Lecture 1

The doubly linked list is a standard non-circularly linked list. It has a head and tail that point to the beginning and end of the list. In a standard doubly linked list, each node is connected to the node before and after it. There are no dummy nodes, and the nodes added by the insert function are added to the top of the list. Below is a diagram of Map.

m\_size = 5

MAP

m\_tail

Pairs

m\_key

m\_value

next

previous

Pairs

m\_key

m\_value

next

previous

Pairs

m\_key

m\_value

next

previous

Pairs

m\_key

m\_value

next

previous

Pairs

m\_key

m\_value

next

previous

m\_head

\\\

If you’re reading this, I appologize as there is suppose to be a nice sized diagram using fancy Microsoft Word boxes here that depict my Map object. I guess I can just describe it. In Map, there are only 3 member variables: two pointers m\_head and m\_tail, and an integer m\_size. m\_head points to the first element of a pair, which has a key, value, a pointer to next, and a pointer to previous. The next pointer points to the next pair, and previous points to the previous. m\_tail points to the last element. Size keeps track of the number of pairs.

Pseudocode:

Map::pairCheck(key)

*Checks if list is empty. If true,returns a null pointer.*

*Repeatedly:*

*Create temp pointer copy of m\_head*

*Compares temp key and parameter key*

*If match, return memory location temp*

*Set temp to the next element after m\_head*

*Return null pointer.*

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

*Checks if list is empty or already contains key. If either, return false.*

*Allocate memory for pair and copy data into it*

*Link it to m\_head location*

*Set m\_head to point to it*

*If applicable, set next element’s previous pointer to it*

*Increment m\_size*

Map::update(const KeyType& key, constValueType& value) (Similar to other simple functions)

*Create temp pointer set to pairCheck(key)*

*Checks if temp is null, if true return false*

*Sets m\_value of temp to value*

Map::erase(const KeyType& key)

*Create temp pointer set to pairCheck(key)*

*Checks if temp is null, if true return false*

*Checks for 3 cases for temp*

*Case 1: temp is only pair in map*

*Do nothing*

*Case 2: temp is first pair in map*

*Set m\_head to next pair*

*Set next elements previous pointer to null pointer*

*Case 3: temp is last pair in map*

*Set m\_tail to previous pair*

*Set previous elements next pointer to null pointer*

*Decrement m\_size*

*Delete temp*

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

*Checks that i is between 0 and m\_size. If not return false.*

*Create temp pointer set to m\_head*

*For value i, temp traverses to the next node i times*

*Copy temp m\_key and m\_value into key and value, respectively*

bool combine(const map& m1, const map& m2, map& result)

*Create empty map*

*Create bool done initialized to true*

*Repeatedly:*

*From m1*

*Get key and value with get function*

*Insert them into temp as pair*

*Repeatedly:*

*From m2*

*Get key and value with get function*

*Insert them into temp as pair*

*Repeatedly:*

*From temp*

*Get key and value with get function*

*If key is in both m1 and m2*

*If value of key in both m1 and m2 are not equal*

*Delete the pair and set done to false*

*Set result to temp*

*Return done*

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

*Copy construct temp with parameter m1*

*Repeatedly:*

*Get key and value from temp*

*If key is in m2*

*Delete the pair*

*Set result to temp*

Test Cases:

(With Keytype as strings and Valuetype as doubles, #include <iostream>, using namespace std;)

Map m;

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

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

assert(m.insert(“Name”, 20)); //Test insert

assert(!m.insert(“Name”, 100)); //Can’t insert, already exists

assert(m.update(“Name”, 10)); //Test update

assert(m.size() == 1); //Size update

assert(!m.empty()); //No longer empty

assert(!m.update(“Tree”, 420)); //No key, can’t update

assert(m.insertOrUpdate(“Fin”, 23)); //Test insertOrUpdate

assert(m.insertOrUpdate(“Tree”, 0)); //Same as above

assert(m.contains(“Name”)); //Test contains

assert(m.contains(“Fin”)); //Same as above

assert(!m.contains(“Me”)); //No key in map

assert(m.erase(“Name”)); //Test delete

assert(!m.erase(“Name”)); //No longer in map

assert(!m.erase(“Fish”)); //Not in map

string s;

double d = 0;

assert(m.get(“Name”, d)); //Test get 2 param

cout << d;

assert(!m.get(“Fame”, d)); //No key, no change

for (int i = 0; i < m.size(); i++)

{assert(m.get(i, s, d)); //Test get 3 param

cout << s << “ “ << d << endl; //Output all pairs

Map m2;

m2.insert(“Ted”, 23);

m2.insert(“Joan”, 99);

m.swap(m2); //Test swap

assert(m.contains(“Ted”) && m.contains(“Joan”) && !m.contains(“Tree”) && !m.contains(“Fin) && !m.contains(“Name”));

assert(m2.contains(“Tree”) && m2.contains(“Fin”) && m2.contains(“Name”) && !m2.contains(“Ted”) && !m2.contains(“Joan”));

Map m3, m4, m5;

m3.insert(“A”, 1);

m3.insert(“B”, 2);

m3.insert(“C”, 3);

m4.insert(“B”, 3);

m4.insert(“C”, 3);

m4.insert(“D”, 4);

assert(!combine(m3, m4, m5)); //Test combine

assert(m5.contains(“A”) && m5.contains(“C”) && m5.contains(“D”) && !m5.contains(“B”));

assert(combine(m3, m3, m3)); //Check for aliasing issues

assert(m3.contains(“A”) && m3.contains(“B”) && m3.contains(“C”));

subtract(m3, m4, m5); //Test subtract

assert(m5.contains(“A”) &&!m5.contains(“B”) && !m5.contains(“C”) && !m5.contains(“D”));

subtract(m3, m3, m5); //Test subtract

assert(m5.empty() && m5.size() == 0);

subtract(m3, m4, m3); //Check for aliasing issues

assert(m3.size() == 1 && m3.contains(“A”);

Map m6;

m6.insert(“Z”, 26);

m6.insert(“Y”, 25);

m6.insert(“X”, 24);

Map m7(m6); //Test Copy Constructor

Assert(m7.contains(“Z”) && m7.contains(“Y”) && m7.contains(“X”));

Map m6(m6); //Will not compile

Map m8;

m8 = m7; //Test Assignment Operator

Assert(m8.contains(“Z”) && m8.contains(“Y”) && m8.contains(“X”));

m8 = m8; //Test Aliasing