CS 32 Project 2 Report

Margaret Capetz

1. I implemented a circular doubly linked list. There is a dummy node that only contains a next and prev pointer and struct Pair that is initialized but never accessed. The list nodes contain a next and prev pointer, as well as a struct Pair which holds a key, value pair. The list nodes are ordered alphanumerically and are sorted upon insertion.

Diagram

Description automatically generatedDiagram

Description automatically generated

1. Pseudocode for non-trivial algorithms

Bool Map::erase(const KeyType& key)

If key is not in map

Return false

Else if key is in map

Create a new node pointer that points to the found key in the map (killMe)

Create a node pointer that points to the node prior to killMe (prevNode)

Create a node pointer that points to the node after killMe (nextNode)

The node after prevNode should be nextNode

The node before nextNode should be prevNode

Dereference killMe pointer (delete)

Decrement linkedlist size

Return true

Diagram

Description automatically generated

Bool Merge(const Map& m1, const Map2 m2, Map& result)

Initailzie bool isMerge = true

Iterate through m1

Set temporary key and value types

Use second get function with index to update temp vars with values

If both m1 and m2 contain the key

Check if both keys have the same corresponding value

If the keys have the same corresponding value

Insert key val pair into map

Else same key, different corresponding values

Set isMerge bool to false

Break

Else if the key appears in exactly 1 of m1 and m2

Insert the key, val pair into result

Iterate through m2

Set temp key and value types

Use index get function to populate the vars

If the key is not in m1 and m2

Get the key and value by index get

Insert pair into result //insert only works if the pair doesn’t exist in list

Return isMerge

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

Check for aliasing by seeing if &m == &result

If both m and result point to the same result

Create a map newResult

If m has only one pair

Insert in result

Return true

If m size is even, switch values between groups of 2

Iterate through map, going two at a time (0, 2, 4, 6…)

Use temp key, val types and index get function to get values for i

Get values for i+1

Insert in newResult pairs where the values of the keys are switched

Else if m size is odd

Hold the first key, value pair in temp variables

Iterate through the list starting at index 1

Get key and val from index get function

Insert in the newResult map a pair of the current key, prev temp value

Set tempVal to current val

After iteration through list, insert in newResult the first key, current temp val

Set the result = newResult;

Return;

Else there is no aliasing

If m has only one pair

Insert in result

Return true

If m size is even, switch values between groups of 2

Iterate through map, going two at a time (0, 2, 4, 6…)

Use temp key, val types and index get function to get values for i

Get values for i+1

Insert in newResult pairs where the values of the keys are switched

Else if m size is odd

Hold the first key, value pair in temp variables

Iterate through the list starting at index 1

Get key and val from index get function

Insert in the newResult map a pair of the current key, prev temp value

Set tempVal to current val

After iteration through list, insert in newResult the first key, current temp val

Diagram, text

Description automatically generated

Copy constructor

Map::Map(const Map &src)

Reset dummy pointers

Dummy->next = dummy;

Dummy->prev = dummy

Copy over new size

Copy over nodes of linked list

Create a node q that points to the next node after dummy

Create a node p that points to the next node after src dummy

While loop to iterate through src list while p != src dummy

New node n

Node after q = n

Node before n = q

Copy p->pair into n->pair

Iterate p: p = p->next

Set q to n

Close the loop

Node after q is dummy

Node before dummy is q

Text, whiteboard

Description automatically generated

Assignment operator

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

If &src = this (aliasing) return \*this and do nothing

Free the memory

Linked list node m = node after dummy

While(m doesn’t equal dummy)

New node pointer k = node after m

Delete m

M = k

Close the list

Dummy->next = dummy;

Dummy->prev = dummy;

Assign new size

Copy over nodes of linked list (same as copy constructor)

Create a node q that points to the next node after dummy

Create a node p that points to the next node after src dummy

While loop to iterate through src list while p != src dummy

New node n

Node after q = n

Node before n = q

Copy p->pair into n->pair

Iterate p: p = p->next

Set q to n

Close the loop

Node after q is dummy

Node before dummy is q

Return statement: return \*this

Bool map::doInsertOrUpdate(const KeyType& key, const ValueType& value)

If key exists in map

Create a node n that points to the key that exists in the map

Update key and val

Else key does not exist in map

Insert in sorted order

Make a node pointer p that points to the node after dummy

While(p != dummy)

If you reach the first node that has a pair key greater than p pair

Break

Else iterate p: p = p->next

Insert

Create new node

Put key and val data in node

Create a node that points to the node before p (prevNode)

Link the nodes so that the order is prevNode -> n -> p

Increment size

Return true

1. Tests were performed on a map from strings to doubles.

**void** test1() { //basic functions

cerr << "test 1 start" << endl;

// default constructor

Map x;

// For an empty map:

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

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

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

Map m;

assert(m.size() == 0); //assert initial size is zero

assert(m.empty() == **true**); //assert the map is empty

m.insert("a" , 1);

m.insert("b" , 2);

m.insert("c" , 3);

assert(m.size() == 3); //assert size is 3 after adding 3 pairs

m.dump();

m.update("a" , 100);

m.dump();

assert(m.size() == 3); //assert size doesn't change after updating

m.erase("b");

assert(m.size() == 2); //assert size decreases after erasing

assert(m.contains("b") == **false**); //assert pair is gone after erase

m.dump();

m.erase("a");

m.erase("c");

assert(m.size() == 0); //assert size is 0 when there are no more pairs

assert(m.contains("a") == **false**); //assert erased pair is not longer in list

assert(m.update("a", 1) == **false**); //pair is no longer in map, update should return false

cerr << "test 1 passed" << endl;

}

**void** test2() { //insert/update functions

cerr << "test 2 start" << endl;

Map m;

m.insert("a" , 1);

m.insert("b" , 2);

m.insert("c" , 3);

assert(m.insert("a", 1) == **false**); //cannot insert if there is already a key existing

ValueType val;

assert(m.get("a", val) == **true** && val == 1); //can get and update val

assert(m.get("z", val) == **false** && val == 1); //cannot get if not in map, val is unchanged

assert(m.update("a", 2) == **true**); //can update

assert(m.get("a", val) == **true** && val == 2); //should fetch updated value

m.dump();

cerr << "test 2 passed" << endl;

}

**void** test3() { //2nd get function

cerr << "test 3 start" << endl;

Map m;

m.insert("z", 1);

m.insert("a", 3);

m.insert("g", 5);

m.insert("b", 4);

m.dump();

KeyType key;

ValueType val;

assert(m.get(0, key, val) == **true** && key == "a" && val == 3); //the key that is strictly greater than 0 keys is the first key which should be "a"

assert(m.get(-1, key, val) == **false** && key == "a" && val == 3); //if i < 0 then false, key/value pair should not be altered

assert(m.get(4, key, val) == **false** && key == "a" && val == 3); //if i >= size then false, key/value pair should not be altered

assert(m.get(1, key, val) == **true** && key == "b" && val == 4); //the key strictly greater than 1 key is "b" the second key

Map msd; // KeyType is std::string, ValueType is double

msd.insert("ccc", 80);

msd.insert("aaa", 40);

msd.insert("ddd", 60);

msd.insert("bbb", 20);

KeyType k = "xxx";

ValueType v;

assert(!msd.get(4, k, v) && k == "xxx"); // x is unchanged

assert(msd.get(1, k, v) && k == "bbb"); // "bbb" is greater than

// exactly 1 item

cerr << "test 3 passed" << endl;

}

**void** test4() {

cerr << "test 4 start" << endl;

Map m;

m.insert("x1", 1);

m.insert("x2", 1);

m.insert("x3", 1);

m.insert("x4", 1);

m.insert("x5", 1);

m.insert("x6", 1);

m.insert("x7", 1);

Map n;

m.swap(n);

assert(n.contains("x1") == **true**); //check that the swap works between an empty map and a filled map

cerr << "m" << endl;

m.dump();

cerr << "n" << endl;

n.dump();

Map l;

l.insert("y1", 2);

l.insert("y2", 2);

l.insert("y3", 2);

n.swap(l);

assert(n.contains("y2") == **true**); //check swap works between non-empty maps

assert(l.contains("x5") == **true**); //check swap works between non-empty maps

cerr << "n" << endl;

n.dump();

cerr << "l" << endl;

l.dump();

cerr << "test 4 passed" << endl;

}

**void** test5() { //merge function

cerr << "test 5 start" << endl;

Map m;

m.insert("fred", 123);

m.insert("ethel", 456);

m.insert("lucy", 789);

Map n;

n.insert("lucy", 789);

n.insert("ricky", 321);

Map o;

o.insert("lucy", 000);

o.insert("ricky", 321);

Map p; //empty

Map result;

assert(merge(m, n, result) == **true**); //merge returns true because it doesn't run into the case in which a key has two unique vals in both maps

// cerr << "result" << endl;

// result.dump();

// cerr << "result end" << endl;

Map result2;

assert(merge(m, o, result2) == **false**); //merge returns false because one key has two vals in both maps

assert(result2.contains("ethel") == **true**);

assert(result2.contains("fred") == **true**);

assert(result2.contains("ricky") == **true**); //should not contain lucy because two lucy keys had 2 different values between the two maps

// cerr << "result 2" << endl;

// result2.dump();

// cerr << "result 2 end" << endl;

Map result3;

assert(merge(m, p, result3) == **true**); //merge returns true because merging a non-empty and empty map should be successful

assert(result3.contains("fred") == **true**);

assert(result3.contains("ethel") == **true**);

assert(result3.contains("lucy") == **true**); //assert that result contains contents of Map m

// cerr << "result 3" << endl;

// result3.dump();

// cerr << "result 3 end" << endl;

Map x;

Map y;

Map result4;

assert(merge(x, y, result4) == **true**);

assert(result4.empty() == **true**);

cerr << "test 5 passed" << endl;

}

**void** test6() {

cerr << "test 6 start" << endl;

//"Fred" 123 "Ethel" 456 "Lucy" 789 "Ricky" 321

Map m;

m.insert("fred", 123);

m.insert("ethel", 456);

m.insert("lucy", 789);

m.insert("ricky", 321); //testing a map that has an even size

Map result;

reassign(m, result);

cerr << "result" << endl;

result.dump();

cerr << "result end" << endl;

ValueType val;

assert(result.get("fred", val) == **true** && val != 123);

assert(result.get("ethel", val) == **true** && val != 456);

assert(result.get("lucy", val) == **true** && val != 789);

assert(result.get("ricky", val) == **true** && val != 321);

Map n;

n.insert("shiv", 123);

n.insert("tom", 456);

n.insert("greg", 789);

reassign(n, result); //testing when result is nonempty

cerr << "result" << endl;

result.dump();

cerr << "result end" << endl;

assert(result.get("shiv", val) == **true** && val != 123);

assert(result.get("tom", val) == **true** && val != 456);

assert(result.get("greg", val) == **true** && val != 789);

Map g;

g.insert("t", 1);

g.insert("k", 2);

g.insert("l", 3);

Map result2;

result2 = g; //assignment operator

reassign(g, result2);

assert(result2.get("t", val) == **true** && val != 1);

assert(result2.get("k", val) == **true** && val != 2);

assert(result2.get("l", val) == **true** && val != 3);

cerr << "before reassign" << endl;

cerr << "m" << endl;

m.dump();

reassign(m, m);

cerr << "after reassign" << endl;

cerr << "m" << endl;

m.dump();

assert(m.get("fred", val) == **true** && val != 123);

assert(m.get("ethel", val) == **true** && val != 456);

assert(m.get("lucy", val) == **true** && val != 789);

assert(m.get("ricky", val) == **true** && val != 321);

Map result3;

Map k;

k.insert("greg", 1); //test size of 1

reassign(k, result3);

assert(result3.contains("greg") && result3.size() == 1);

k.insert("", 78);

k.insert("t", 1);

k.insert("k", 2);

k.insert("l", 3);

k.insert("fred", 123);

k.insert("ethel", 456);

k.insert("lucy", 789);

k.insert("ricky", 321); //testing a larger list

reassign(k, result3);

assert(result3.get("t", val) == **true** && val != 1);

assert(result3.get("k", val) == **true** && val != 2);

assert(result3.get("l", val) == **true** && val != 3);

assert(result3.get("fred", val) == **true** && val != 123);

assert(result3.get("ethel", val) == **true** && val != 456);

assert(result3.get("lucy", val) == **true** && val != 789);

assert(result3.get("ricky", val) == **true** && val != 321);

assert(result3.get("", val) == **true** && val != 78);

cerr << "test 6 passed" << endl;

}

**void** test7() { //test aliasing for merge

cerr << "test 7 start" << endl;

Map m;

Map n;

Map result;

merge(m, n, result);

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

m.insert("kendall", 12);

m.insert("randall", 345);

m.insert("roman", 355);

merge(m, m, result);

assert(result.contains("kendall") && result.contains("randall") && result.contains("roman"));

n.insert("shiv", 88);

assert(merge(m, n, result));

result.dump();

assert(merge(m, n, m));

m.dump(); //should still function the same as above, despite aliasing

assert(merge(m, n, result));

n.insert("roman", 0);

assert(merge(m, n, result) == **false**); //returns false bc two different vals for key type "roman"

cerr << "n" << endl;

assert(merge(n, n, n));

n.dump();

cerr << "test 7 passed" << endl;

}