Kalin Khemka

304336969

Report Project 2

**Description of Doubly-Linked List**

The doubly-linked lists used in this project deal with data nodes, all of which contain the key, the value, the pointer to the next data in the list, and the pointer to the previous data in the list. The list is not necessarily circular because I did not include a tail pointer, but you can traverse it forwards and backwards. I have head pointer pointing to the first node in the list. There is no specific order to the list. Every time a node is inserted, it is simply inserted at the beginning of the list, which means that the head pointer changes every time a new node is added.

Head pointer

Key 4

Key 3

Key 2

Key 1

nullptr

nullptr

Value 4

Value 3

Value 2

Value 1

**Pseudocode**

pseudocode for non-trivial algorithms (e.g., Map::erase and subtract).

Private:

data\* find(const KeyType& key) const;

set a new pointer to the node pointed to by m\_head

repeatedly

checks key parameter same as node’s key

returns the node where keys are the same

cycles to next node

if none found, returns nullptr

public:

Map(const Map& other);

Sets the length of new list

Creates new pointer to create new list

Sets the head pointer to node

Creates new pointer to cycle through other list

Repeatedly

Checks pointer to other list is nullptr

Copies over keys and values if not

Checks to see if other list has a nextData

If it does, then makes a new node for new list

Sets the previous pointer in that new node

Checks to see if the previous pointer in other list’s node is nullptr

Sets the new list’s node’s previous pointer to nullptr if so

Moves to next pointer within both lists

Map& operator= (const Map& other);

Checks to make sure that other map isn’t same as this map

If not, creates a copy of other map

Swaps the copy with this map

Returns pointer to this map

~Map();

Creates new pointer to the head node

Checks if pointer pointing to nullptr

If it is, does nothing

Repeatedly

Checks if the next node is a nullptr

If not, moves to next node and deletes the previous

Deletes this last node

bool contains(const KeyType& key) const;

Sets a new pointer equal to find(key)

Returns true if not a nullptr

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

finds node with key

returns false if not nullptr

Creates a new node and sets key and value

Sets the next data to the node that the head pointer points to

The previous data is set to nullptr

Sets the the next node’s previous pointer to this new node if possible

Updates the head pointer to point to this node

Increases length of list

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

finds node with key

if the pointer is nullptr

return false

updates the value of the node

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

finds node with key

Checks for a nullptr

Inserts the key and value

Otherwise updates value of the node

bool erase(const KeyType& key);

finds node with key

if pointer is nullptr return false

Checks if the next data is a nullptr, if not

Checks if last data is a nullptr, if not

Changes the last pointer of next node to the last pointer of current node

Changes the next pointer of last node to next pointer of current node

Otherwise sets head pointer to the next pointer of current node

Otherwise checks if the last pointer is nullptr and then makes the last pointers next pointer point to null

Delets the node, decreases length

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

finds node with key

returns false if nullptr

sets value parameter to value of node

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

Checks whether the i is within bounds

Cycles through list to find node i

Sets parameters to the key and value of node

void swap(Map& other);

creates temp node

swaps head pointers to “swap” the maps

creates temp length

swaps the lengths of maps

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

creates temp map

Cycles through first list

Cycles through second list

Checks if the keys are same at any point in both lists (changes flag)

Inserts the key from list one to temp list if not duplicated

Cycles through second list

Cycles through first list

Checks if the keys are same at any point in both lists(changes flag)

Inserts the key from list two to temp list if not duplicated

Sets result map to temp map

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

creates temp map

Cycles through second list

Cycles through first list

Checks if the keys are same at any point

Inserts key from m1 to temp list if not duplicated

Sets result map to temp map

**Test Cases**

The tests were performed on a map from strings to doubles:

Map a, b;

KeyType k = "Fred", k2 = "Bob", k3 = "George";

ValueType v = 123, v2 = 345, v3 = 567;

assert(a.empty()); //Checking empty

assert(b.empty());//Checking empty

assert(a.insert(k,v)); //Inserting

assert(a.insert(k2,v2));//Inserting

assert(!a.insert(k2,v));//Checking if can double insert

assert(a.size() == 2);//Checking size

assert(b.insert(k3,v3));//Inserting

assert(!b.insert(k3,v3));//Checking double insert

assert(b.size() == 1); //Checking size

assert(a.contains(k)); //Checking contains function

assert(!b.contains(k));//Checking contains function

KeyType k4,k5;

ValueType v4, v5;

assert(a.update(k,v2)); //Checking update

assert(a.get(k,v4) && v4 == v2); //Making sure value updated

assert(a.update(k,v)); //Resetting value

v4 = 0; //Making sure next commands won't be a fluke

assert(a.get(0,k4,v4) && ((k4 == "Fred" && v4 == 123) || (k4 == "Bob" && v4 == 345))); //Checking to see if the values are in fact in the map

assert(a.get(1,k5,v5) && ((k5 == "Fred" && v5 == 123) || (k5 == "Bob" && v5 == 345))); //Checking to see if the values are in fact in the map

KeyType k6,k7;

ValueType v6, v7;

assert(b.insertOrUpdate(k,v)); //Checking insertOrUpdate (inserting)

assert(b.contains(k));//Checking to see if added

assert(!b.contains(k2));//Making sure it doesn't have a value not added

assert(b.get(k3,v6) && v6 == v3);//Checking to make sure the value is in fact there

assert(b.insertOrUpdate(k3,v2)); //Updating that value

assert(b.get(k3,v6) && v6 == v2); //Checking whether it updated

assert(b.update(k3,v3)); //Resetting it

assert(b.get(k3,v7) && v7 == v3); //Making sure it reset

assert(b.erase(k)); //Checking erase function

assert(!b.contains(k)); //Making sure it erased

assert(b.size() == 1); //Checking the size to make sure

a.swap(b); //Swapping the linked lists

assert(a.size() == 1); //Making sure size changed

assert(b.size() == 2); //Making sure size changed

assert(!a.contains(k)); //Checking to see if the values switched

assert(b.contains(k)); //Checking to see if the values switched

assert(b.contains(k2)); //Checking to see if the values switched

assert(a.contains(k3)); //Checking to see if the values switched

Map c = b;//Essentially Map a from the beginning - Copy Constructor Test

Map d = a;//Essentially Map b from the beginning - Copy Constructor Test

assert(c.contains(k)); //Checking to see if the values were copied

assert(!d.contains(k)); //Checking to see if the values were copied

assert(c.contains(k2)); //Checking to see if the values were copied

assert(d.contains(k3)); //Checking to see if the values were copied

Map e;

assert(combine(c,d,e)); //Checking combine function

assert(e.contains(k) && e.contains(k2) && e.contains(k3) && e.size() == 3); //Checking to see if the new map has the combined values

Map f;

subtract(c,d,f); //Checking subtract

assert(f.contains(k) && f.contains(k2) && f.size() == 2); //Making sure the resultant list has the correct values

Map g;

assert(g.insert(k3,v3)); //Making a new map and adding a value to it

g = f; //Assignment Operator Test

assert(g.contains(k) && g.contains(k2) && g.size() == 2); //Seeing if the assignment operator works

assert(combine(c,e,f)); //Combining with a non-empty resultant map passed in as third value

assert(f.contains(k) && f.contains(k2) && f.contains(k3) && f.size() == 3); //Making sure the old f is overwritten by the combined list

assert(c.update(k,v3)); //Changing a value in Map c

assert(!combine(c,e,f)); //Seeing if combine will return false now that the same key has 2 different values in 2 different maps

assert(f.contains(k2) && f.contains(k3) && f.size() == 2); //Checking the resultant map to make sure it's correct

assert(c.update(k,v)); //Resetting the value

assert(c.insert(k3,v3)); //Adding the third key to the Map

subtract(c,d,g); //Checking subtract with a non-empty resultant map passed into the third value

assert(g.contains(k) && g.contains(k2) && g.size() == 2); //Checking to see if that old g was overwritten with the new subtracted map