#include "Map.h"

#include <iostream>

**using** **namespace** std;

//default constructor for Map

Map::Map()

{

//set head and tail pointing to nothing, and the size of the map to 0

head = **nullptr**;

n = 0;

tail = **nullptr**; //idk if i need a tail yet

}

//copy constructor

Map::Map(**const** Map& src)

{

//if there is nothing in the source map, we use default settings

**if**(src.n == 0) {

head = **nullptr**;

tail = **nullptr**;

n = 0;

}

//if there is only one item (head) in src, we initialize just head, and set tail to nothing

**else** **if** (src.n == 1) {

head = **new** Node;

head->previous = **nullptr**;

head->next = **nullptr**;

head->key = src.head->key;

head->value = src.head->value;

tail = **nullptr**;

n = 1;

} **else** {

//this means there is more than one item

//first set the head to proper values and pointers

Node\* iterator = src.head;

head = **new** Node;

head->previous = **nullptr**;

head->key = iterator->key;

head->value = iterator->value;

Node\* newiterator = head;

//loop through the rest of src and link each new node with next and previous

**for**(**int** i = 1; i < src.n; i++) {

iterator = iterator->next;

Node\* add = **new** Node;

add->key = iterator->key;

add->value = iterator->value;

add->previous = newiterator;

newiterator->next = add;

newiterator = newiterator->next;

}

//final tail stuff because it gets cut off

tail = newiterator;

tail->next = **nullptr**;

//set the size of map

n = src.n;

}

}

Map::~Map()

{

//do nothing if the map is empty - there is nothing to be deconstructed

**if**(n==0) {

**return**;

}

//if the size is one we just delete head

**if**(n==1) {

**delete** head;

**return**;

}

//otherwise, we loop through starting from head to just before tail, deleting everything

Node\* kill = head;

Node\* next=kill->next;

**for**(**int** i = 1; i < n-1; i++) {

**delete** kill;

next = next->next;

kill=next->previous;

}

//finally, delete the last two items

**delete** kill;

**delete** next;

}

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

{

**if**(&src == **this**) {

**return** \***this**;

}

// dump();

// src.dump();

//need to take down the original map. This is the same code as the deconstructor.

**if**(n==0) {

//do nothing

}

**else** **if**(n==1) {

**delete** head;

} **else** {

Node\* kill = head;

Node\* next=kill->next;

**for**(**int** i = 1; i < n-1; i++) {

**delete** kill;

next = next->next;

kill=next->previous;

}

**delete** kill;

**delete** next;

}

//now I have to assign stuff. This is the same code as the copy constructor since I'm just copying values into an empty map

//see copy constructor for more commented code

//if there is nothing in the source map

**if**(src.n == 0) {

head = **nullptr**;

tail = **nullptr**;

n = 0;

}

//if there is only one item (head) in src

**else** **if** (src.n == 1) {

head = **new** Node;

head->previous = **nullptr**;

head->next = **nullptr**;

head->key = src.head->key;

head->value = src.head->value;

tail = **nullptr**;

n = 1;

} **else** {

Node\* iterator = src.head;

head = **new** Node;

head->previous = **nullptr**;

head->key = iterator->key;

head->value = iterator->value;

Node\* newiterator = head;

**for**(**int** i = 1; i < src.n; i++) {

iterator = iterator->next;

Node\* add = **new** Node;

add->key = iterator->key;

add->value = iterator->value;

add->previous = newiterator;

newiterator->next = add;

newiterator = newiterator->next;

}

//final tail stuff

tail = newiterator;

tail->next = **nullptr**;

n = src.n;

}

**return** \***this**;

}

//check if the map is empty

**bool** Map::empty() **const**

{

**return** n==0;

}

//get the size of the map

**int** Map::size() **const**

{

**return** n;

}

//insert key and value if key is not already in the map

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

{

//edge case: map is empty and we just create a head node with the key and value

**if**(head == **nullptr**) {

head = **new** Node;

head->next = **nullptr**;

head->previous = **nullptr**;

head->value = value;

head->key = key;

n++;

**return** **true**;

}

//If the function reaches here, the map is not empty

//return false at any time if the key trying to be inserted is the same as a key already in the map

Node\* iterator = head;

**if** (iterator->key == key) {

**return** **false**;

}

**while**(iterator->next != **nullptr**)

{

// cerr << "key in insert check: " << iterator->key << endl;

// cerr << "key to check against: " << key << endl;

//return false if the keys are the same

**if** (iterator->key == key) {

**return** **false**;

}

iterator = iterator->next;

}

//checks the tail

**if** (iterator->key == key) {

**return** **false**;

}

//if it reaches here, no keys are the same and we make a new node to add to the end, upding the nexts and previous

Node\* add = **new** Node;

add->key = key;

add->value = value;

iterator->next = add;

add->previous = iterator;

add->next = **nullptr**;

//update the tail pointer

tail = add;

//update the size of the map

n++;

**return** **true**;

}

//update a key, value pair to a new value if the key is in the map

**bool** Map::update(**const** KeyType& key, **const** ValueType& value)

{

Node\* iterator = head;

//cycles through the entire map, checking each node to see if the keys are the same. If so, update the value

**while**(iterator != **nullptr**) {

**if**(iterator->key == key) {

iterator->value = value;

**return** **true**;

}

iterator = iterator->next;

}

//if it reaches here, no keys are the same and the map remains the same. Returns false because doesn't update.

**return** **false**;

}

//either inserts or updates based on the key, value pair given.

**bool** Map::insertOrUpdate(**const** KeyType& key, **const** ValueType& value)

{

//edge case: empty map. Calls insert function to insert.

**if**(head == **nullptr**) {

insert(key, value);

**return** **true**;

}

//check to see if we can update, and return true if updates

**if**(update(key, value)) {

**return** **true**;

}

//otherwise we just insert because nothing is updated

insert(key, value);

**return** **true**;

}

//gets rid of a node with keyvalue key, and returns false if it doesn't contain the key

**bool** Map::erase(**const** KeyType& key)

{

//if there is only one item in map and head has the key, then we just get rid of head and set everything to nullptr, as well

//as decrement the size

**if**(head != **nullptr** && head->key == key && n==1) {

**delete** head;

head=**nullptr**;

tail=**nullptr**;

n--;

**return** **true**;

}

//cycles through the map

Node\* iterator = head;

**while**(iterator != **nullptr**) {

//if we find the key, then we have to get rid of that node.

**if**(iterator->key == key) {

//if head is the one to get rid of

**if**(iterator->key == head->key) {

//set a new head

iterator = iterator->next;

iterator->previous = **nullptr**;

**delete** head;

head = iterator;

n--;

//update tail pointer

**if**(n == 1) {

tail = **nullptr**;

}

**return** **true**;

}

//other edge case: the last one is the one to get rid of

**else** **if** (iterator->next == **nullptr**) {

Node\* previous = iterator->previous;

previous->next = **nullptr**;

**delete** iterator;

n--;

//update tail

tail = previous;

**return** **true**;

}

//everything in the middle

**else** {

Node\* previous = iterator->previous;

Node\* next = iterator->next;

previous->next = next;

next->previous = previous;

**delete** iterator;

n--;

**return** **true**;

}

}

iterator = iterator->next;

}

//if it gets here, nothing has been deleted since nothing matches. Return false.

**return** **false**;

}

//check if the map contains a key

**bool** Map::contains(**const** KeyType& key) **const**

{

//loops through the map

Node\* iterator = head;

**while**(iterator != **nullptr**)

{

//returns true if finds the key

**if**(iterator->key == key)

{

**return** **true**;

}

iterator = iterator->next;

}

//didn't find the key, return false

**return** **false**;

}

//gets and stores the value of key in value and returns true; if it fails, return false

**bool** Map::get(**const** KeyType& key, ValueType& value) **const**

{

//cycles through the map

Node\* iterator = head;

**while**(iterator != **nullptr**) {

//sets value and returns true if it finds the key

**if**(iterator->key == key) {

value = iterator->value;

**return** **true**;

}

iterator = iterator->next;

}

//returns false if it didn't find the key

**return** **false**;

}

//gets the key and value at index i, returns false if i is not in the proper bounds.

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

{

//i isn't within bounds, return false

**if**(i < 0 || i >= size()) {

**return** **false**;

}

//loop through until i is reached

Node\* iterator = head;

**for**(**int** j = 0; j < i; j++) {

iterator = iterator->next;

}

//set key and value to the key and value of the node at i

key = iterator->key;

value = iterator->value;

**return** **true**;

}

//swaps the map with other

**void** Map::swap(Map& other)

{

//temp map is copy constructed to this map's object

Map temp = \***this**;

//this object is now assigned to other with the reassignment operator

\***this** = other;

//other is now reassigned to temp, which is identical to the original this

other = temp;

}

//just a dump function for my own use

**void** Map::dump() **const**

{

cerr << "size: " << size() << endl;

cerr << "Empty?: " << empty() << endl;

Node\* iterator=head;

**while**(iterator != **nullptr**) {

cerr << "Key: " << iterator->key << " Value: " << iterator->value << endl;

iterator = iterator->next;

}

**if**(head != **nullptr**)

cerr << "head key: " << head->key << " Head value: " << head->value << endl;

**if**(tail != **nullptr**)

cerr << "tail key: " << tail->key << " tail value: " << tail->value << endl;

}

//combines m1 and m2 into result

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

{

//the return value of the function

**bool** retval = **true**;

Map m;

m = m1;

**for**(**int** i = 0; i < m2.size(); i++) {

//get the key and value of m2

KeyType m2key;

ValueType m2value;

m2.get(i, m2key, m2value);

// cerr << "m2key: " << m2key << " m2val: " << m2value << endl;

//if m1 doesn't contain key, insert it into result

**if**(!m.contains(m2key)) {

// cerr << "doesnt contain " << m2key << endl;

m.insert(m2key, m2value);

} **else** {

//get the value of m1 for the key if the m1 contains the same key as m2

ValueType m1val;

m.get(m2key, m1val);

// cerr << "m2key: " << m2key << " m1val: " << m1val << endl;

**if**(m1val != m2value) {

//if the two values are not the same, we get rid of the node containing the key and set the return value to false

m.erase(m2key);

retval = **false**;

}

//otherwise, it does no insertions or deletions since m1 already contains the right key and value pair

}

}

result = m;

//returns the boolean return value that is false if at any point two keys were the same but their values were different

//otherwise, it's still true

**return** retval;

}

//reassigns values in m to different keys, and stores the result in result

**void** reassign(**const** Map& m, Map& result)

{

//set result to be m (assignment operator)

result = m;

//if the size of m is 0 or 1, it does nothing

**if**(m.size() == 0 || m.size() == 1) {

**return**;

}

//otherwise, we have temporary keys and temporary values

KeyType key1;

KeyType key2;

ValueType temp1;

ValueType temp2;

//gets key and value at head

result.get(0, key1, temp1);

//gets the key at head for later

KeyType headKey = key1;

//loops through the result map and udpates each successive key until the end with the previous value

**for**(**int** i = 1; i < result.size(); i++) {

result.get(i, key2, temp2);

result.update(key2, temp1);

temp1 = temp2;

key1 = key2;

}

//head is the only one not updated, so update head with the old last value (tail value)

result.update(headKey, temp1);

}

//

// Map.hpp

// Project2

//

// Created by Christopher Clark on 1/23/20.

// Copyright © 2020 Christopher Clark. All rights reserved.

//

#ifndef Map\_h

#define Map\_h

#include <stdio.h>

#include <string>

**using** KeyType = std::string;

**using** ValueType = **double**;

**class** Map

{

**public**:

Map();

~Map();

Map &**operator**=(**const** Map& src);

Map(**const** Map& src);

**bool** empty() **const**;

**int** size() **const**;

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

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

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

**bool** erase(**const** KeyType& key);

**bool** contains(**const** KeyType& key) **const**;

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

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

**void** swap(Map& other);

**void** dump() **const**;

**private**:

**int** n;

**struct** Node

{

Node\* previous;

Node\* next;

KeyType key;

ValueType value;

};

Node \*head;

Node \*tail;

};

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

**void** reassign(**const** Map& m, Map& result);

#endif