CS32 Project 2 Report

Description:

I constructed my doubly-linked list with a class map that contains a head and tail pointer, which are pointers to nodes, and a record of the number of nodes in the map. Each node in the map contains data of a Keytype (specified to std::string) and Valuetype (specified to double), a next pointer, and prev pointer. The next pointer points to the next node in the map, the prev pointer points to the previous node in the map. Whenever the map’s constructor is called, its head and tail pointers will be set to nullptr and the number of nodes it contains (stored in private member variable int m\_numNodes) will be initialized to zero. Whenever a new node is added to the list, it will be added to the end of the list and the tail pointer will be updated. My implementations for the functions insert and insertOrUpdate ensures there will be no duplicate keys in the map.

Pseudocodes:

Private Member Funtions:

void addToFront(const KeyType& key, const ValueType& value)

- dynamically allocate a new node

- set its m\_key and m\_value to its corresponding values passed in from the parameter

- initialize its next and prev node pointer to nullptr

- if adding to an empty map,

- set head and tail to the dynamically allocated node

- increment number of nodes in map

- if adding to a non-empty map

- connect the dynamically allocated node to the head

- update head’s position to the dynamically allocated node

- increment number of nodes in map

void addToRear(const KeyType& key, const ValueType& value)

- dynamically allocate a new node

- set its m\_key and m\_value to its corresponding values passed in from the parameter

- connect the dynamically allocated node to the tail

- update tail to the dynamically allocated node

- increment number of nodes in map

Public Member Functions (Non-Trivial):

bool erase(const KeyType& key);

- check if map is empty and whether target key is in the map

- if map is empty or does not contain target key, return false

- if target node is first node in map

- create a node that points to the head

- if target node is the only node in map

- set head and tail to nullptr

- delete the node

- decrement number of nodes in map

- if target node is the first node in map, and map has more than one nodes

- update head’s position to node after head

- delete target node

- decrement number of nodes in map

- if instead, target node is in somewhere middle of the map of end of the map

- allocate a pointer to head

- traverse through map to find target node

- if target node is in the middle

- allocate a pointer to target node

- reset the next and prev pointer of the nodes before and after target node so they are connected to each other

- delete target node

- decrement number of nodes in map

- if target node is the last node

- allocate a pointer to target node

- set the next pointer of the node before to nullptr

- set tail to the node before

- delete target node

- decrement number of nodes in map

- return true after target node has been deleted

void swap(Map& other);

- initialize 2 temporary node pointers to store the head and tail pointers

- initialize a variable to store number of nodes in map

- store head, tail, and number of nodes of current map in the temporary pointers and variable

- set current map’s head and tail pointers to the other map’s head and tail pointers, do the same for the other map but with current map’s head and tail pointers

- set current map’s size to the other map’s size, do the same for the other map but with current map’s size

Map (const Map &oldMap); //copy constructor

- allocate a temp node that points to the oldMap’s head

- if the oldMap is empty

- create an empty map by setting head and tail to nullptr

- set number of nodes to zero

- if oldMap is not empty

- dynamically allocate a new node

- set its key and value to that of oldMap’s head

- set its next and prev pointers to nullptr

- set head and tail to this new node

- move the temp node to next node in the oldMap

- calls insert (which will add new node to current map)

- keep repeating last two steps until temp node has traversed through oldMap

Map &operator= (const Map &oldMap); //assignment operator

- check if oldMap and current map are referring to the same map

- if oldMap and current map are the same, return current map

- allocate a node to current map’s head

- traverse though and delete each node of current map

- set current map’s head and tail pointer to nullptr

- allocate a temp node that points to the oldMap’s head

- if the oldMap is empty

- create an empty map by setting head and tail to nullptr

- set number of nodes to zero

- if oldMap is not empty

- dynamically allocate a new node

- set its key and value to that of oldMap’s head

- set its next and prev pointers to nullptr

- set head and tail to this new node

- move the temp node to next node in the oldMap

- calls insert (which will add new node to current map)

- keep repeating last two steps until temp node has traversed through oldMap

Map Algorithms (Non-Trivial):

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

- create an empty map (tempMap)

- if m1 and m2 are the same map (aliasing)

- calls assignment operator make empty map a deep copy of m1

- set result to tempMap

- set result to tempMap, which have all data of m1

- if m1 and m2 are not the same

- declare and initialize a boolean value (diffval) that is used to check whether keys present in both m1 and m2 have the same value

- loop through m1 and get key and value of each node (through get with 3 parameters)

- insert node with the key and value only if it is not contained in m2 (check with the function contains)

- if a key is also contained in m2

- calls get with 2 parameters to check if values of the keys are same

- only insert into tempMap if the values are the same

- if not the same, mark this difference with the boolean value (diffval)

- loop through m2 and get key and value of each node (through get with 3 parameters)

- if a key is not contained in m1, insert into tempMap

- set result to tempMap

- return true or false depending on the boolean value (diffval)

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

- create an empty map (temp)

- if m1 and m2 are the same, set result to this empty map

- if m1 and m2 are not the same,

- set temp as m1

- loop through m2

- if temp contains a key in m2, erase the key

- set result to temp

Test Cases:

Map m; //calls default constructor

// testing for insert

assert(m.insert("steph", 100) && m.insert("beep", 20)&& m.insert("hi", -11));

assert(!m.insert("steph",-8)); //key "steph" should not be inserted since it's already in map m

/\*

Map m should contain the following key value pairs:

steph 100

beep 20

hi -11

\*/

//test for contains and size

assert(m.contains("steph") && m.contains("beep") && m.contains("hi"));

assert(m.size()==3);

//these 2 assertions should return true

KeyType k;

ValueType v;

//test for insertOrUpdate

Map m1;

assert(m1.insert("account", 21) && m1.insertOrUpdate("gene block", 4.1) && m1.insertOrUpdate("steph",100));

assert(m1.insertOrUpdate("steph", 99)); //key "steph"'s value should be updated to 99 by insertOrUpdate

assert(m1.get("steph",v)); //(test for get function that takes 2 parameters)

assert(v== 99); //v should be the value of the key "steph", which is 99

assert(m1.contains("steph") && m1.contains("gene block") && m1.contains("account"));

assert(m1.size()==3);

/\*

Map m1 should contain the following key value pairs:

account 21

gene block 4.1

steph 99

\*/

assert(m1.get(-1,k,v)==false); //3 parameters get function should return false for passing a negative index

assert(m1.get(200,k,v)==false); //3 parameters get function should return false for passing an index larger than size of map

assert(m1.get(3,k,v)==false);//3 parameters get function should return false for passing an index that is equal to size of the map

assert(m1.get(1,k,v)); //3 parameters get function should return true for values from 0~size()-1

assert(k == "account" || k == "steph" || k == "gene block"); //the get function should have retrieved one of the keys in m1

assert(v == 21 || v == 4.1 || v == 99); //the get function should have retrieved one of the values in m1

//testing for erase

Map m2;

assert(m2.insert("haha", 2) && m2.insertOrUpdate("ha", 1) && m2.update("ha", -5));

assert(m2.erase("ha") && !m2.erase("steph")); //cannot erase a key not contained in m2

assert(m2.erase("haha")); //erasing the key "haha" in m2

assert(m2.empty()); //should result in an empty map m2

assert(m2.size()==0); //an empty map should have size 0

assert(m2.get(0,k,v) == false); //an empty map should not contain any key

Map empty; //calls default constructor to construct empty maps

Map result;

ValueType v1 = 0;

//test for map algorithms (combine and subtract)

assert(combine(m,m1,empty)==false); //combine should return false because the values for the key "steph", which are present in both m and m, are not equal

assert(empty.contains("beep") && empty.contains("hi") && empty.contains("account") && empty.contains("gene block")); //this assertion should return true

subtract(m,m1,result);

assert(result.contains("beep") && result.contains("hi")); //result should contain all keys only in m but not m1, which are beep and hi

assert(result.get("beep",v) && result.get("hi", v1));

assert(v == 20 && v1 == -11); //the keys in result should have the same corresponding values as before

m1.update("steph",100); //test for update, value for the key "steph" in m1 should now be 100

assert(combine(m,m1,empty)==true && empty.contains("steph")); //values for the key "steph" are now same in both map, so combine should return true and map "empty" should now also contain the key "steph"

//tests for map algorithms with aliasing and when third parameter is not an empty map

subtract(m, m1, m); //result should contain "beep" and “hi”, same as when the third parameter is an empty map

//since the third parameter is same as the first parameter, the key "steph" is now erased from m

assert(m.contains("beep") && m.contains("hi")); //result should contain all keys only in m but not m1, which are beep and hi

subtract(m1, m1, result);//passed same map for the first and second parameter

assert(result.empty()); //all keys are the same in the same map, so result should be empty

subtract(m,m2,m2); //passed same map for second and third parameter, but result should contain all keys present in m regardless

assert(m2.contains("beep") && m.contains("hi"));

Map empty2; //constructs an empty map

subtract (empty2, m1, result); //result should contain an empty map

assert(result.empty());

combine(m,m,m); //pass same map for all 3 parameters, m should not be changed

assert(m.contains("beep") && m.contains("hi"));

subtract(m,m,m); //pass same map for all 3 parameters, m should now be an empty map

assert(m.empty());

/\*

Map m should now contain the following key value pairs:

steph 100

beep 20

hi -11

Map m1 should contain the following key value pairs:

account 21

gene block 4.1

steph 100

Map m2 should now contain the following key value pairs:

beep 20

hi -11

Map empty should now contain the following key value pairs:

steph 100

beep 20

hi -11

account 21

gene block 4.1

\*/

//test for copy constructor

Map m3 = empty;

assert(m3.contains("steph") && m3.contains("beep") && m3.contains("hi") && m3.contains("account") && m3.contains("gene block"));

//the new map m3 should contain all key/value pairs of the map "empty"

//test for assignment operator

m = m1;

assert(m.contains("steph") && m.contains("gene block") && m.contains("account"));

empty = empty2; //test for passing an empty map to the assignment operator

assert(empty.empty());

/\*

Map m should now contain the following key value pairs:

account 21

gene block 4.1

steph 100

Map m1 should contain the following key value pairs:

account 21

gene block 4.1

steph 100

Map m2 should now contain the following key value pairs:

beep 20

hi -11

Map m3 should now contain the following key value pairs:

steph 100

beep 20

hi -11

account 21

gene block 4.1

and Map empty is now empty

\*/

//test for swap

empty.swap(empty2); //swapping two empty map results in an empty map

m1.swap(m3);

m2.swap(empty2); //swapping a non-empty map with an empty map

assert(m2.empty());

assert(m1.size()== 5 && m3.size() == 3); //the two maps should now have each other's sizes

assert(m1.contains("beep") && m1.contains("hi")); //m1 now has content of m3

assert(!m3.contains("beep")&& !m3.contains("hi")); //m3 lost contents that were not present in m1