CS 32 Project 2 Report

* Arsh Malik

Description of Linked List used:

The linked list implemented in this project is a **non-circular** doubly-linked list **without** dummy nodes. A node of the linked list has 4 items – the key, the value, the **next** pointer pointing to the next node and the **prev** pointer pointing to the previous node. The following image is a sample linked list.

A picture containing text, electronics

Description automatically generated

Pseudocode for non-trivial functions:

1. **Copy constructor**

Map::Map(const Map& other)

check if the Map passed as parameter is empty

then the current map will be empty

otherwise

repeatedly loop through the parameter map

get the key value pairs of that map

insert it into the current map

1. **Assignment operator**

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

if the current map is different from rhs

create a temp copy of rhs

swap the current map with temp

return pointer to this map

1. **Insert function**

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

if the map already contains the key

return false

if the map is empty

create a new node and set its key and values same as parameters of function

set next and prev pointers to nullptr

connect the head and tail pointers to this node

increment size

// pre sorting while inserting

if the key is the smallest

create a new node and set its key and values same as parameters of function

link the next pointer to the original first node

link the prev pointer to nullptr

set head to point to the new node

increment size

return true

repeatedly loop through the map till the last node(not go past)

if the node after the current node points to nullptr

break

otherwise, if the key param is greater than the current node’s key and less than the next node’s key

// new Node should be created here

create a new node and set its key and value equal to parameters.

connect the pointers to the previous and next nodes

increment the size

return true

if the key parameter is less than the last node’s key

// need to make a new node in the second last position

create a new node and set its key and value equal to parameters.

connect the pointers to the previous and next nodes

increment the size

return true

take p to the last node

create a new Node and set its key and value equal to the parameters.

set the next pointer to nullptr and prev to p

set tail to new node

set last element’s next to new node

increment size

return true

1. **Update function**

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

repeatedly, iterating through the linked list

if a node’s key matches key parameter

set its value to value parameter and return true

return false

1. **Erase function**

bool Map::erase(const KeyType& key)

if the map doesn’t contain the key

then return false

if the first node is the node to be deleted

if the next pointer of node pointed by head is not nullptr:

set a pointer to a Node equal to head

set head to point to the Node after the node to be deleted

the node next to the node to be deleted should have prev = nullptr.

delete the node pointed by toDelete

decrement m\_size

return true

else

set head, tail to nullptr

decrement size and delete node;

repeatedly, iterating through the linked list

if the key of the node(p) match the parameter

set the prev of node after the node to be deleted to

point to p’s previous node.

set p’s previous node’s next point to p’s next node

delete p, decrement m\_size

and return true

//deleting the last node

set the tail pointer point to the second last node

set the next pointer of second last node to nullptr

delete the last node, decrement m\_size

and return true

1. **Get function with 3 parameters**

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

if i is out of bounds

return false

initialize a counter

set a pointer p to the head of the map

repeatedly iterate through the map and set p to the ith node

get the key and value of the ith node and set the reference parameters.

return true

1. **Swap function**

void Map::swap(Map& other)

swap the head pointers using a temporary pointer to Node variable

swap the tail pointers using a temporary pointer to Node variable

swap the size private member using a temporary integer variable

1. **Merge function**

bool merge(const Map& m1, const Map& m2, Map& result)

create a temp map and assign m1 to it

initialize a bool var returnVal to true

repeatedly iterate through m2 using int i

get the ith key and value

if m1 contains the same key

get the value of that key in m1

if the value doesn’t match the value of key in m2

remove the k,v pair from temp

set returnVal to false

otherwise,

insert the ith key and value in temp

finally set result equal to temp

return returnVal

1. **Reassign function**

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

assign m to result

get the first node’s key and value of result

repeatedly, starting from the second node to iterate through m

getting the current node’s value and store in local variable

get the previous node’s key and store in local variable

assigning that value to the previous node

get the key of the last node

set the last node’s key map to first node’s value

1. **Destructor**

Loop through the linked list

create a new node pointer and point it to the following node

delete the current node

Test Cases:

The tests were performed on a map from strings to doubles

|  |  |
| --- | --- |
| Test Case | Reason to use |
| Map m;  assert(m.size()==0);  assert(m.insert("Fred", 123));  assert(m.insert("Ethel", 456));  assert(m.size() == 2);  Map m1;  m1 = m;  assert(m1.contains("Fred") && m1.contains("Ethel") && m1.size()==2); | Ensuring default constructor is working.  The insert function works.  Check the size function  Check the work of copy constructor. |
| Map m1;  Map m2 = m1;  assert(!m2.contains("Ethel") && m2.size()==0); | To check if the copy constructor works when the source map is empty. |
| Map m;  assert(m.insert("Fred", 123));  assert(m.insert("Ethel", 456));  assert(m.size() == 2);  Map m1;  m1.insert("John", 789);  m1.insert("Akira", 890);  m1.insert("Tim", 234);  m1 = m;  assert(!m1.contains("John") && !m1.contains("Akira") && !m1.contains("Tim"));  assert(m1.contains("Fred") && m1.contains("Ethel") && m1.size()==2); | To check the working of the assignment operator when rhs is not same as lhs. |
| Map m;  assert(m.insert("Fred", 123));  assert(m.insert("Ethel", 456));  assert(m.size() == 2);  m = m;  assert(m.contains("Fred"));  assert(m.contains("Ethel"));  assert(m.size() == 2); | Assignment operator should work correctly when lhs = rhs, returning pointer to the same object. |
| Map m;  assert(m.empty()==true);  assert(m.insert("Fred", 123));  assert(m.insert("Ethel", 456));  assert(m.size() == 2);  assert(m.empty()==false); | the empty function should return true when there is no node in the map and false when nodes have been inserted |
| Map m;  assert(m.empty()==true);  assert(m.size()==0);  assert(m.insert("Fred", 123));  assert(m.insert("Ethel", 456));  assert(m.size() == 2);  assert(m.empty()==false); | Testing the size function- It should return 0 if map is empty otherwise an integer that is a count of the number of items in the map. |
| Map m;  assert(m.empty()==true);  assert(m.contains("halo") == false);  assert(m.size()==0);  assert(m.insert("Fred", 123));  assert(m.insert("Ethel", 456));  assert(m.contains("Fred") && m.contains("Ethel"));  assert(m.size() == 2);  assert(m.empty()==false); | Checking the **contains** function. For empty map, it should return false and for non-empty map, it should look for elements and return true. If not found, return false. |
| Map m;  assert(m.insert("Fred", 123));  assert(m.insert("Ethel", 456));  assert(m.contains("Fred") && m.contains("Ethel"));  assert(m.insert("Fred", 0) == false); | Insert function should return false if the key already exists in the map. |
| Map m;  assert(m.insert("Fred", 123));  assert(m.contains("Fred") && m.size()==1); | First key-value pair should be inserted successfully in a map. |
| Map m;  assert(m.insert("b", 123));  assert(m.insert("c", 234));  assert(m.insert("a", 345));  KeyType k; ValueType v;  assert(m.get(0, k, v) && k == "a" && v == 345); | If the key that needs to be inserted is less than the current first key, then it should be inserted in the front. |
| Map m;  assert(m.insert("a", 345));  assert(m.insert("c", 123));  assert(m.insert("g", 234));  assert(m.insert("x", 1000));  KeyType k; ValueType v;  assert(m.get(1, k, v) && k == "c" && v == 123);  assert(m.insert("b", 5) && m.size() == 5);  assert(m.get(1, k, v) && k == "b" && v == 5);  assert(m.get(4, k, v) && k == "x" && v == 1000);  assert(m.insert("h", 1) && m.size()==6);  assert(m.get(4, k, v) && k == "h" && v == 1);  m.dump(); | If we try to insert the keys somewhere in the middle of the linked list, then they should be inserted successfully and order of the linked list must be ascending.  **Output:**  a 345  b 5  c 123  g 234  h 1  x 1000  Passed all tests  Program ended with exit code: 0 |
| Map m;  assert(m.insert("a", 345));  assert(m.insert("c", 123));  assert(m.insert("g", 234));  assert(m.insert("x", 1000));  KeyType k; ValueType v;  assert(m.get(1, k, v) && k == "c" && v == 123);  assert(m.insert("b", 5) && m.size() == 5);  assert(m.get(1, k, v) && k == "b" && v == 5);  assert(m.get(4, k, v) && k == "x" && v == 1000);  assert(m.insert("h", 1) && m.size()==6);  assert(m.get(4, k, v) && k == "h" && v == 1);  assert(m.insert("z", 999) && m.size()==7);  assert(m.get(6, k, v) && k == "z" && v == 999);  m.dump(); | Checking if the key is inserted correctly if it is greater than all the other keys. It should be place in the end of the linked list. |
| Map m;  assert(m.insert("a", 345));  assert(m.insert("c", 123));  assert(m.insert("g", 234));  assert(m.insert("x", 1000));  KeyType k; ValueType v;  assert(m.update("c", 100));  assert(m.get("c",v) && v == 100);  assert(!m.update("z", 0)); | Update the value if key is found and return true else return false. |
| Map m;  assert(m.insert("a", 345));  assert(m.insert("c", 123));  assert(m.insert("g", 234));  assert(m.insert("x", 1000));  KeyType k; ValueType v;  assert(m.insertOrUpdate("c", 100));  assert(m.get("c",v) && v == 100);  assert(m.insertOrUpdate("z", 0)); | InsertOrUpdate should always return true. If the key is found, update its value. Otherwise, insert that key with the passed value. |
| Map m;  assert(m.insert("a", 123));  assert(m.insert("b", 234));  assert(m.insert("c", 345));  assert(m.erase("c") == true);  assert(m.contains("c") == false); | If the key is found, then erase the k,v pair from the map. Erasing the last key value pair. |
| Map m;  assert(m.insert("a", 123));  assert(m.insert("b", 234));  assert(m.insert("c", 345));  assert(m.erase("z") == false); | If the key is not in the map then erase should return false. |
| Map m;  assert(m.insert("a", 123));  assert(m.insert("b", 234));  assert(m.insert("c", 345));  assert(m.erase("a") == true &&m.size()==2);  assert(!m.contains("a")); | Erasing the first node |
| Map m;  assert(m.insert("a", 123));  assert(m.insert("b", 234));  assert(m.insert("c", 345));  assert(m.erase("b") == true && m.size()==2);  assert(!m.contains("b"));  m.dump(); | Erasing a node in the middle of a linked list |
| Map m;  assert(m.insert("a", 123));  assert(m.insert("b", 234));  assert(m.insert("c", 345));  assert(m.erase("c") == true && m.size()==2);  assert(!m.contains("c"));  m.dump(); | Erasing the last node |
| Map m;  assert(m.insert("a", 123));  assert(m.erase("a"));  assert(m.size()==0);  m.dump(); | Erasing the only node from the map |
| Map m;  ValueType v;  assert(m.get("a", v) == false); | No value to get in an empty map |
| Map m;  m.insert("a", 1);  m.insert("b", 2);  m.insert("c", 3);  ValueType v;  assert(m.get("a", v) == true && v == 1); | Getting the value of a key that exists in the map. |
| Map m;  m.insert("a", 1);  KeyType k; ValueType v;  assert(m.get(0, k, v) == true && k=="a" && v == 1); | Getting key,value from the single node linked list |
| Map m;  m.insert("a", 1);  m.insert("k", 2);  m.insert("c", 3);  m.insert("x", 4);  m.dump();  KeyType k; ValueType v;  assert(m.get(0, k, v) == true && k=="a" && v == 1);  assert(m.get(1, k, v) == true && k == "c" && v == 3);  assert(m.get(2, k, v) == true && k == "k" && v == 2);  assert(m.get(3, k, v) == true && k == "x" && v == 4); | The get function should set the k and v variables appropriatly.  While inserting, the map is sorted in ascending order. |
| Map m;  KeyType k; ValueType v;  assert(m.get(9, k, v) == false);  m.insert("a", 1);  m.insert("k", 2);  m.insert("c", 3);  m.insert("x", 4);  m.dump();  assert(m.get(-1, k, v) == false);  assert(m.get(4, k, v) == false); | If i is out of range or map is empty before getting, get should return false |
| Map m1;  Map m2;  m1.insert("A", 1);  m1.insert("B", 2);  m2.insert("X", 9);  m1.swap(m2);  assert(m1.size() == 1 && m1.contains("X"));  assert(m2.size() == 2 && m2.contains("A") && m2.contains("B")); | Swap function should swap the key value pairs and the size of both the maps would change. |
| Map m1;  Map m2;  m1.insert("A", 1);  m1.insert("B", 2);  m1.swap(m2);  assert(m1.size() == 0 && !m1.contains("A"));  assert(m2.size() == 2 && m2.contains("A") && m2.contains("B")); | Swapping should work if one map is empty |
| Map m1;  Map m2;  m1.swap(m2);  assert(m1.size() == 0 && !m1.contains("A"));  assert(m2.size() == 0 && !m2.contains("A") && !m2.contains("B")); | Swap should also work when both the maps are empty. |
| Map m1;  Map m2;  m1.insert("Fred", 123);  m1.insert("Ethel", 456);  m1.insert("Lucy", 789);  m2.insert("Lucy", 56);  m2.insert("Ricky", 321);  Map result;  assert(!merge(m1, m2, result));  result.dump(); | If there exists a key that appears in both m1 and m2, but with different corresponding values, then this function returns false. |
| Map m1;  Map m2;  m1.insert("Fred", 123);  m1.insert("Ethel", 456);  m1.insert("Lucy", 789);  m2.insert("Lucy", 789);  m2.insert("Ricky", 321);  Map result;  assert(merge(m1, m2, result));  result.dump(); | If a key appears in exactly one of m1 and m2, then result must contain a pair consisting of that key and its corresponding value. |
| Map m1;  Map m2;  m1.insert("Fred", 123);  m1.insert("Ethel", 456);  m1.insert("Lucy", 789);  m2.insert("Lucy", 789);  m2.insert("Ricky", 321);  Map result;  assert(merge(m1, m2, m1));  m1.dump(); | Checking if merge is working for aliases. |
| Map m1;  Map m2;  m1.insert("Fred", 123);  m1.insert("Ethel", 456);  m1.insert("Lucy", 789);  m2.insert("Lucy", 789);  m2.insert("Ricky", 321);  Map result;  reassign(m1, m2);  m2.dump(); | Checking if the reassign function works when the result is not empty. |
| Map m1;  Map m2;  m1.insert("Fred", 123);  m1.insert("Ethel", 456);  m1.insert("Lucy", 789);  m2.insert("Lucy", 789);  m2.insert("Ricky", 321);  Map result;  reassign(m1, result);  result.dump(); | Checking if the reassign function works when the result is empty. |
| Map m1;  m1.insert("Fred", 123);  m1.insert("Ethel", 456);  m1.insert("Lucy", 456);  Map result;  reassign(m1, result);  result.dump(); | Check the case when 2 out of 3 elements have the same value. |