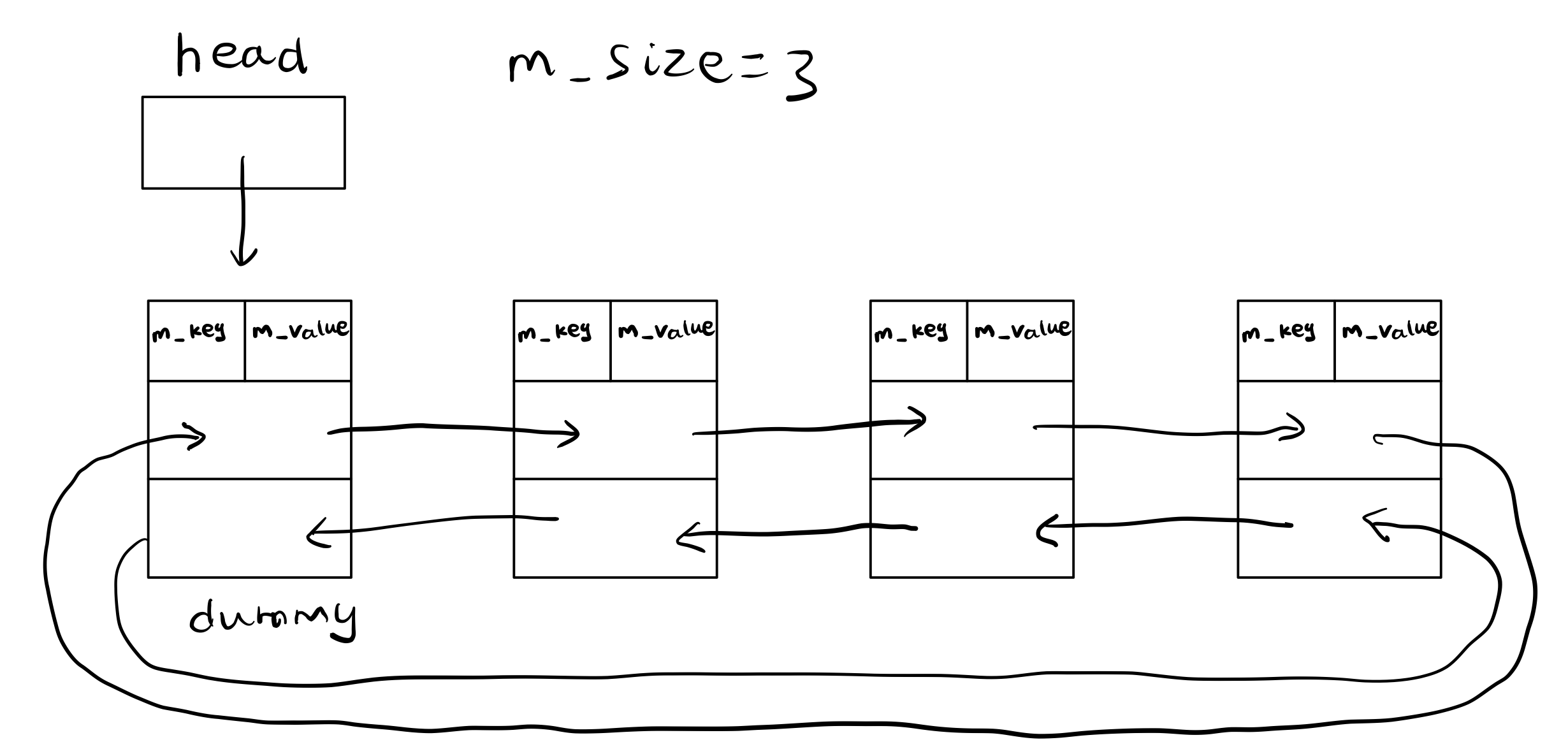
Project 2 report

Wenjie Mo 905304148

1. Design

One map class contains: number of nodes in the Map (m\_size), the head pointer which points to a dummy node, several nodes which contains the pointer to the previous object, the pointer points to the next object, and the value-key pair.

I designed a circular doubly linked list. As shown in the Fig. 1, when the Map is first initialized, the dummy node’s previous and next pointer all point to itself, and the head pointer points to the dummy node. When an element is added into the Map (Fig. 2), it will be added to the last position of the Map: the previous last node’s next pointer will point to this new element and the new pointer’s next pointer will point to the dummy node (the tail node is dummy->m\_prev). The circular doubly linked list gives me the opportunity to access the previous element and the next element of a node, which help the implement of insert and erase function (Fig. 3).

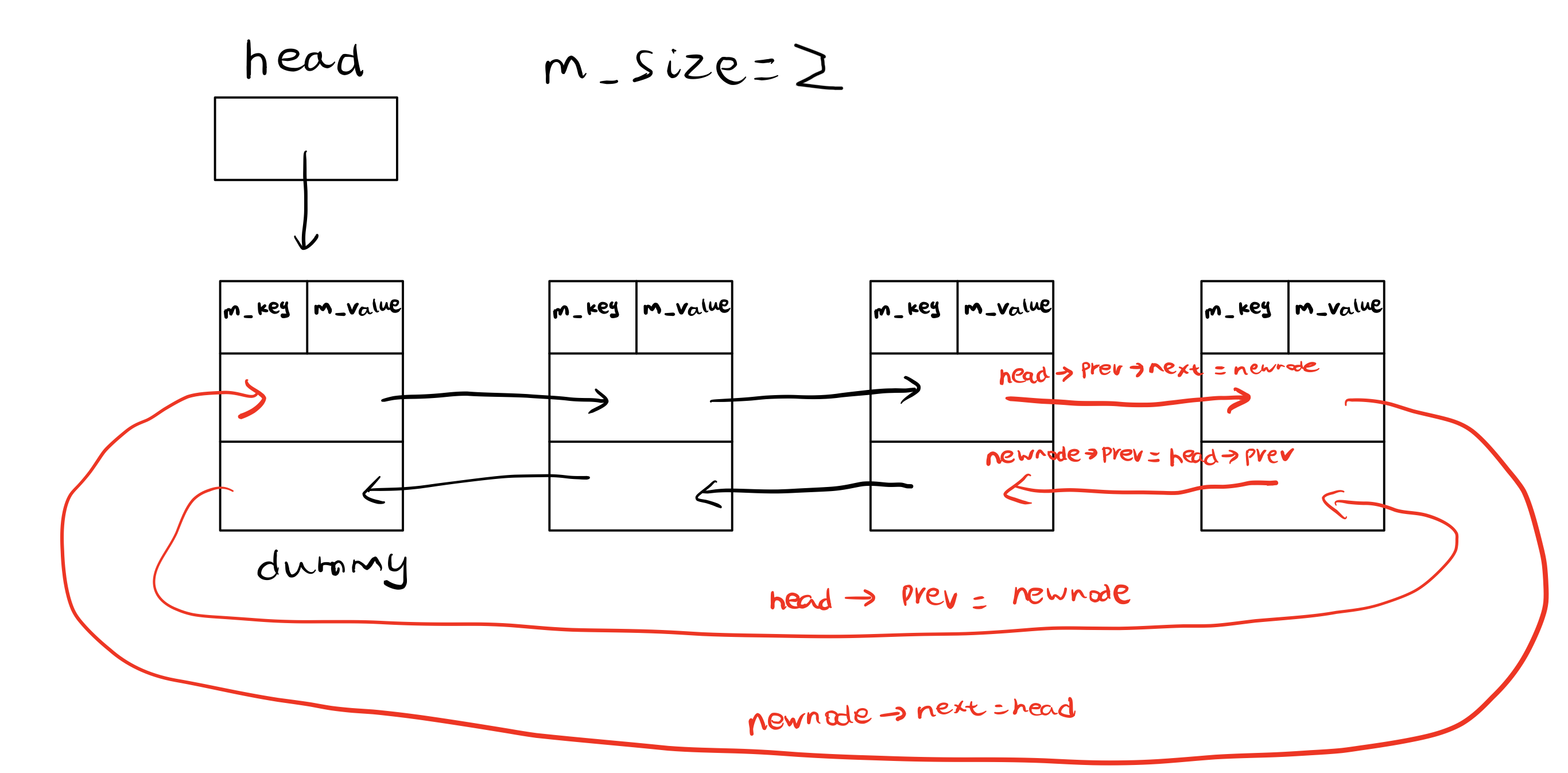
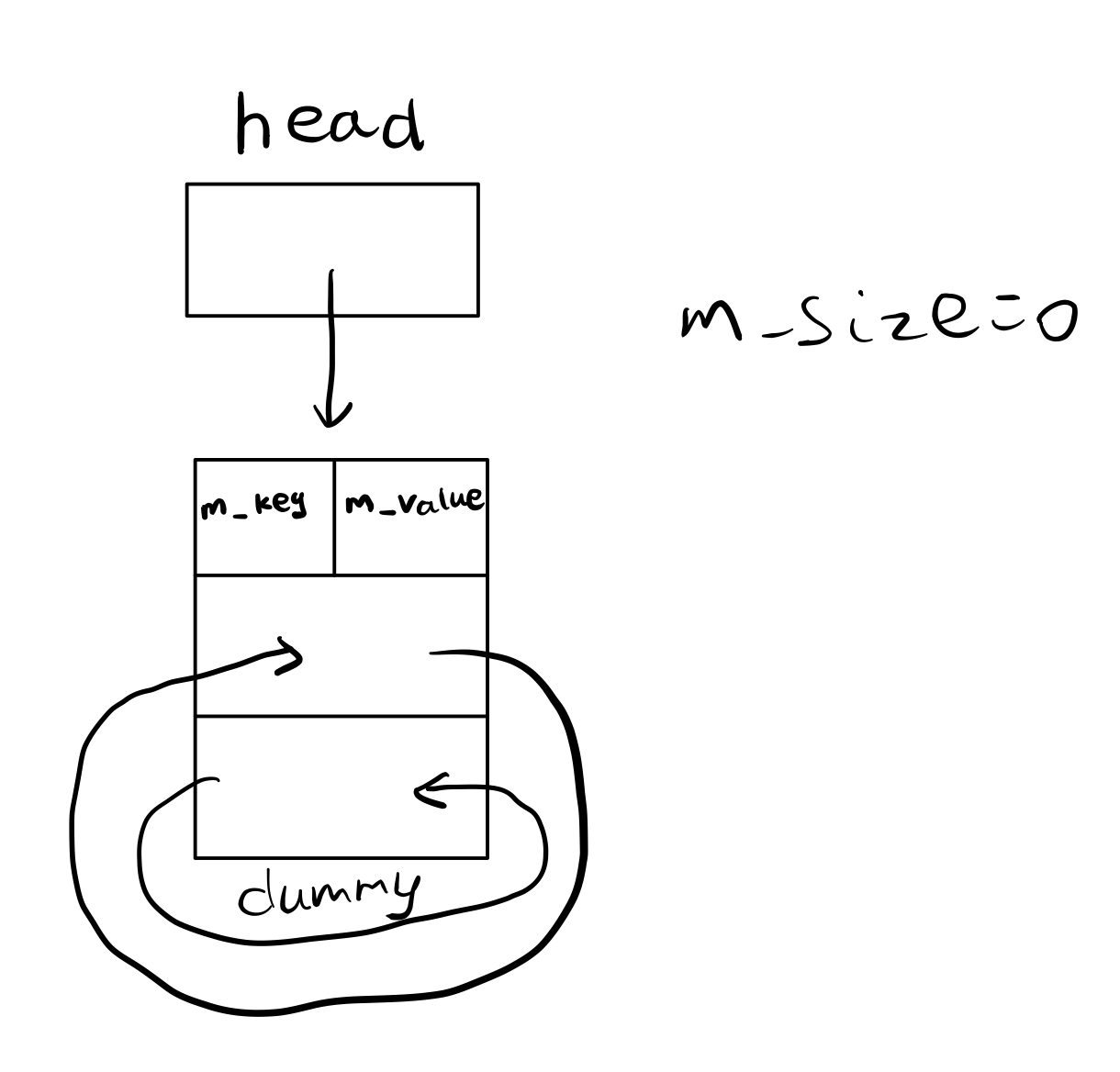
Fig. 1 empty list Fig. 2 add an element

Fig. 3 normal list

2. Pseudocode

Map::Map()

set size to zero

create a new dummy node

assign the dummy node's next and previous pointer is it self

set up a head points to dummy node

Map::~Map()

create a pointer points to the first element

repeat when the pointer does not meet the end of the Map

set up a temporary pointer points to the next element

delete the current values

set the pointer points to the next element

delete the dummy node

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

If the left side does not points to the same Map as right

create a copy of right hand side Map

swap the value of the left hand side with the copy's value (via swap function)

when go out of the scope, the temp with the Map the left hand side Map points to is deleted

return the Map object

Map::Map(**const** Map& other)

set the size to 0

create a new Map with new dummy node

Repeatedly get value&key from this’s node until the end

assign the values in the other Map into the new map

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

if the key is already in the Map

return false

create a new node with values given in the parameter

assign the new node's next pointer to the head

assign the new node's next pointer to the original head's previous pointer (new pointes to the second last node)

assign the original last node's next pointer to the new node/

assign head's previous pointer to new node

size +1

return true

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

if could update (via update function)

insert with given parameter then return true

if could not update, then insert (via update function)

insert with given parameter then return true

If either fails, return false (not possible)

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

repeat until the end of the Map is met

if one key is equal to the requested key

this key's next node's previous node points to its previous node

this key's previous node's next node points to its next node

delete this node, decrease the size, and return true

If the node is not found, return false

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

repeat until the end of the Map is met

if one key is equal to the requested key

return true

if the end is reached without finding the key

return false

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

repeat until the end of the Map is met

if one key is equal to the requested key

set the value parameter to key's value

return true

if the end is reached without finding the key

return false

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

if i is out of bounds, return false

use a loop to reach the ith node

set the key parameter to this node's key

set the value parameter to this node's value

return true

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

create a pointer points to this node's head

assign this node's head points to other's head

assign other's node's head to the original this node's head via the temp pointer

create the temp size

assign this's size to other's size

assign other's size to this's size via temp size

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

use the assignment operator to copy all m1 nodes into result (erase result's original data)

set up a boolean value for return

repeat until the end of the Map m2 is met

get each key-value pair in the m2

insert the value into result

if the key already exists and two values for the same key is different

remove this key from result

return value will be false

if the key already exists, but the values are the same, update the value

return the Boolean value

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

use the assignment operator to copy all m's nodes into result (erase result's original data)

Store the value from first element

starts with the second node repeat until the end of the Map result is met

assign the current key the value the previous node's value

assign the last key the first value

3. Test case

#include "Map.h"

#include <string>

#include <iostream>

#include <cassert>

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

**void** test()

{

Map m;

Map result;

ValueType value1;

KeyType key1 = "Fred";

ValueType value2;

KeyType key2 = "Ethel";

ValueType value3;

KeyType key3 = "Martin";

ValueType value4;

KeyType key4 = "NONE";

assert(m.empty()); // empty

assert(m.insert("Fred", 2.7) && m.insert("Ethel", 3.5)); // test insert function

assert(m.insertOrUpdate("Martin", 3.8) && m.insertOrUpdate("Ethel", 3.2)); // test insertOrUpdate function (insert update)

assert( m.update("Fred", 2.9)); // test the update function

assert(! m.update("Moe", 3.8)); // not exist, cannot update

assert(! m.insert("Martin", 3.7)); // already exist, cannot insert

assert(m.size() == 3 && ! m.empty()); // not empty and size = 3

assert(m.contains("Fred") && m.contains("Ethel") && m.contains("Martin")); // test the contain function

assert(m.insert("Alen", 3.0));

assert(m.erase("Alen") && ! m.erase("AlenAlias") && m.size() == 3); // test the erase function

Map m2(m); // test the copy constructor

assert(m2.size() == 3 && m2.contains("Fred") && m2. contains("Ethel") && m2.contains("Martin"));

assert(m2.get(key1, value1));

assert(m2.get(key2, value2));

assert(m2.get(key3, value3));

assert(value1 == 2.9 && value2 == 3.2 && value3 == 3.8);

assert(! m2.get(key4, value4)); // retrieve value of a not existing key

assert(! m2.get(4, key4, value4)); // i out of bonds

assert(! m2.get(-1, key4, value4)); // i out of bonds

assert(m2.get(0, key4, value4) && (key4 == "Fred" || key4 == "Ethel" || key4 == "Martin")); // First element in m2

value1 = 0;

value2 = 0;

value3 = 0; // reassign the value

Map m3;

assert(m3.insert("Fred", 2.9) && m3.insert("Bla", 3.5));

Map m4(m3); // copy constructor

assert(m4.size() == 2 && m4.contains("Fred") && m4. contains("Bla")); // check whether the copy is successful

m3 = m2; // assignment operator

assert(m3.size() == 3 && m3.contains("Fred") && m3. contains("Ethel") && m3.contains("Martin")); // check whether the copy is successful

m3.get(key1, value1);

m3.get(key2, value2);

m3.get(key3, value3);

assert(value1 == 2.9 && value2 == 3.2 && value3 == 3.8); // check whether the value-key pair is correct

assert(combine(m, m4, result)); // test combine

assert(result.contains("Fred") && result.contains("Ethel") && result.contains("Martin") && result.contains("Bla") && result.size() == 4);

Map m5;

assert(m5.insert("Fred", 2.8) && m5.insert("Bla", 3.5));

assert(! combine(m, m5, result)); // "fred" with different value, the function returns false

assert(! result.contains("Fred") && result.contains("Ethel") && result.contains("Martin") && result.contains("Bla") && result.size() == 3); // Fred should not be in the result, but bla should stay in the result

Map resultNotempty;

assert(resultNotempty.insert("Fake", 2.8)); // special case when result is not empty

assert(combine(m, m4, resultNotempty));

assert(! resultNotempty.contains("Fake") && resultNotempty.size() == 4);

reassign(m2, resultNotempty); // test reassign

assert(resultNotempty.size() == 3 && resultNotempty.contains("Fred") && resultNotempty.contains("Ethel") && resultNotempty.contains("Martin"));

assert(resultNotempty.get(key1, value1));

assert(resultNotempty.get(key2, value2));

assert(resultNotempty.get(key3, value3));

assert(value1 != 2.9 && value2 != 3.2 && value3 != 3.8); // test whether values have been changed

Map m6;

assert(m6.insert("Martin", 3.9)); // test reassign with one element

reassign(m6, resultNotempty);

Map\* m7 = &m2; // test reassign with same pointer

reassign(\*m7,m2);

assert(m2.size() == 3 && m2.contains("Fred") && m2.contains("Ethel") && m2.contains("Martin"));

assert(m2.get(key1, value1));

assert(m2.get(key2, value2));

assert(m2.get(key3, value3));

assert(value1 != 2.9 && value2 != 3.2 && value3 != 3.8); // test whether values have been changed

m7 = &m; // test when two parameter all points to the same Map

Map\* resultPtr = &m;

assert(! combine(m5, \*m7, \*resultPtr));

assert(resultPtr->size() == 3 && ! resultPtr->contains("Fred") && resultPtr->contains("Ethel") && resultPtr->contains("Martin"));

Map\* m8 = &m2; // test when three parameter all points to the same Map

m7 = &m2;

assert(combine(\*m8, \*m7, \*resultPtr));

assert(resultPtr->size() == 3 && resultPtr->contains("Fred") && resultPtr->contains("Ethel") && resultPtr->contains("Martin"));

**int** main()

{

test();

cout << "Passed all tests" << endl;

}