Report

1. description of the design of doubly-linked list implementation
2. it is a circular list, which means each node can not only contain the KeyType and ValueType, but also points to the previous node and next node; the head of the circular list is a dummy node, which means I won’t initialize the KeyType and ValueType of my head. My list node is in a particular oder: they are sorted from the smallest KeyType to the largest KeyType.
3. Here is my diagram of the designed circular list:

Empty map:

Diagram

Description automatically generated

Map which size is two:

Diagram

Description automatically generated

1. pseudocode for non-trivial algorithms
2. bool Map::insert(const KeyType &key, const ValueType &value)

check if the map is empty:

if yes

link the first new node with the nead

refresh the number of element in the map

if no

check if the node’s key has already been existed in the map

if yes

return false

check throughout the map elements

if the new key value is larger than all the other element’s key (which means the next node will be the head)

linked the last element with this new node and link this new node with head (which means the new node’s previous is the last node and the new node’s next is the head)

refresh the number of elements in the circular linked list

return true

else if the new node’s key is larger than the map’s node

link the new node with the map’s node (which means the new node’s previous node is the map’s previous node and the new node’s next node is the map’s next node)

refresh the number of elements in the circular linked list

return true

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

for all the node (except the head node in the map)

if the Key of the node is equal to key

update the Value of the node with value

return true;

if no Key of the node in the map is equal to key

return false

1. bool Map::erase(const KeyType &key)

for all the node(except the head node in the map)

if the Key of the node is equal to key

let the node’s previous node’s next node link to the node’s next node

let the node’s next node’s previous node link to the node’s previous node

delete the node

refresh the number of node in the map

return true

if there is no Key of node in the map equals to key

return false

1. bool Map::contains(const KeyType &key) const

for all the node(except the head node in the map)

if the Key of the node is equal to key

return true

if there is no Key of node in the map equals to key

return false

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

for all the node(except the head node in the map)

if the Key of the node is equal to key

set the value equal to the node’s value

return true

if there is no Key of node in the map equals to key

return false

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

if i is smaller than 0 or is greater and equal to map’s size

return false

else

start to count node from the beginning of map

if count is equal to i

set the key equal to the node’s key at this point

set the value equal to the node’s value at this point

return true

else, continue to count

1. void Map::swap(Map &other)

let the map’s size equal the other’s size

let the other’s size equal to the map’s size

set the map’s head node to the other’s head node

set the other’s head node equal to the map’s head node

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

for all node in result except the head node

erase every node

set bool to true

for every node in m1

if m2 also contains the node’s key

if the m2’s node’ value is equal to that of m1’s

insert that node into result

else

keep looping and set bool to false

else

insert that node into result

for every node in m2

if m1 doesn’t contain the node’s key

insert that node into result

return the bool

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

for all node in result except the head node

erase every node

if there is only one element in m (except the head node)

copy the node and insert it into result

else

loop through all node in m

if the node is the first node

copy the end node’s key and value into that first node

else

copy the previous node’s key and value into this node

1. test cases:

Map a;

assert(a.size() == 0); //a is empty

assert(!a.contains("a")); //a won't contain anything

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

assert(a.insertOrUpdate("a", 1)); //insert a,1 into a

assert(!a.empty()); //now a is not empty

assert(!a.insert("a", 2)); // a has already contains a,1

assert(a.insert("b", 2)); //insert b,2 into a

assert(a.contains("b")); //a now contains b

assert(a.insert("c", 4)); //insert c,4 into a

assert(a.update("c", 3)); //update c with 3 in a

assert(a.insert("d", 4)); //insert d,4 into a

assert(!a.contains("e")); // a does not contain e

assert(a.contains("c")); //a contains c

assert(a.size() == 4); //a's size is 4

ValueType val;

assert(!a.get("f", val)); // a cannot get f

assert(a.get("c", val)); //a can get c with 3

assert(val == 3); //c is corresponding to 3

assert(!a.get("g", val)); // a does not contain g

assert(val == 3); //val doesn't change;

KeyType key;

assert(!a.get(4, key, val)); // a only contains 4 elements

assert(val == 3); //val doesn't change

assert(!a.get(-1, key, val)); //get cannot access to negative number of elements

assert(a.get(0, key, val));

assert(key == "a" && val == 1); //get the a,1

assert(a.get(1, key, val));

assert(key == "b" && val == 2); //get the b,2

assert(a.get(2, key, val));

assert(key == "c" && val == 3); //get the c,3

assert(a.get(3, key, val));

assert(key == "d" && val == 4); //get the d,4

Map k(a); //use copy constuctor to copy k

assert(!k.get(4, key, val)); // k only contains 4 elements

assert(val == 4); //val doesn't change

assert(!k.get(-1, key, val)); //get cannot access to negative number of elements

assert(k.get(0, key, val));

assert(key == "a" && val == 1); //get the a,1

assert(k.get(1, key, val));

assert(key == "b" && val == 2); //get the b,2

assert(k.get(2, key, val));

assert(key == "c" && val == 3); //get the c,3

assert(k.get(3, key, val));

assert(key == "d" && val == 4); //get the d,4

assert(!k.insert("a", 9)); //a,1 is already in the k

assert(k.insert("e", 5)); // insert e,5 into k

assert(k.size() == 5); //now k has 5 elements;

a = k;

assert(a.size() == 5); //a should have the same number of elements as k

assert(!a.get(5, key, val)); // get cannot get access to equal number of size of the map

assert(a.get(1, key, val));

assert(key == "b" && val == 2); //a should still contains b,2

assert(a.get("c", val)); // a should be able to get c

assert(val == 3); //c,3 is in a

assert(a.erase("e")); //erase e from a

assert(a.size() == 4); // not only 4 elements in a

assert(k.size() == 5); //k should not be modified

Map i;

i.insert("a", 10);

i.insert("b", 20);

i.insert("c", 30);

i.insert("d", 40);

i.insert("e", 50);

Map j;

j.insert("f", 60);

j.insert("g", 70);

j.insert("h", 80);

j.insert("a", 30);

Map q;

assert(!merge(i, j, q)); //unable to merge because of the different value of a corresponding to

assert(q.size() == 7); //now q doesn't contain a

assert(!q.contains("a")); //q doesn't contains a

assert(j.erase("a"));

assert(merge(i, j, q)); //able to merge i and j

assert(q.size() == 8); // now every elemtns in i and j is in q

assert(i.insert("i", 90));

assert(j.insert("i", 90));

assert(merge(i, j, q)); //able to merge i and j

assert(q.size() == 9); //q contains element from a to i

assert(q.get(8, key, val));

assert(key == "i" && val == 90); // the largest element in q should be i,90

Map result;

reassign(q, result); // reassign q into result

assert(q.get("a", val));

assert(val == 10); // q doesn't changed

assert(result.get("a", val));

assert(val != 10); // result has been reassigned

assert(q.get("b", val));

assert(val == 20); // q doesn't changed

assert(result.get("b", val));

assert(val != 20); // result has been reassigned

assert(q.get("c", val));

assert(val == 30); // q doesn't changed

assert(result.get("c", val));

assert(val != 30); // result has been reassigned

assert(q.get("d", val));

assert(val == 40); // q doesn't changed

assert(result.get("d", val));

assert(val != 40); // result has been reassigned

assert(q.get("e", val));

assert(val == 50); // q doesn't changed

assert(result.get("e", val));

assert(val != 50); // result has been reassigned

assert(q.get("f", val));

assert(val == 60); // q doesn't changed

assert(result.get("f", val));

assert(val != 60); // result has been reassigned

assert(q.get("g", val));

assert(val == 70); // q doesn't changed

assert(result.get("g", val));

assert(val != 70); // result has been reassigned

assert(q.get("h", val));

assert(val == 80); // q doesn't changed

assert(result.get("h", val));

assert(val != 80); // result has been reassigned

assert(q.get("i", val));

assert(val == 90); // q doesn't changed

assert(result.get("i", val));

assert(val != 90); // result has been reassigned

cerr << "Passed all tests!" << endl;