1. In my project, I implemented a doubly-linked list with a separate head and tail pointer attached to the front and end of my lnked list. When a map is first created, meaning map is empty, the head and tail pointers are both set to nullptr and my m\_size private variable, which records the size of the linked list, is set to zero. For this project, each node consists of a mapValue variable, which holds the key and value, a next pointer, and a previous pointer. Therefore, when a typical map is created each node’s next pointer points to the next node in the linked list and the previous pointer points to the previous node in the linked list. Not to mention, the head pointer would consistently be pointing to the node at the top of the linked list and the tail pointer would consistently be pointing to the node at the bottom of the linked list. The nodes are not in any particular order in the linked list but, when new nodes are inserted, they are added to the bottom of the linked list, so technically, it’s ordered by time of insertion.
2. **PSUEDOCODE**

* Constructor:

set head and tail to nullptr

size equals zero

* Destructor:

create a new pointer and set that pointer equal to head

iterate through linked list

new pointer to node follows to the next node in list

delete current node

follow new pointer to next node

set head and tail to nullptr, and size equals 0

* Copy Constructor:

if copy’s head node is nullptr

set head, tail to nullptr, and size equals zero

copy size

create a new head pointer, setting previous to nullptr

head pointer’s m\_map value equals copy head’s m\_map value

create two new pointers, one pointing to head and another pointing to copy’s head

while copyNode’s next pointer is not nullptr

copy over contents from original node to new copyNode

create a pointer to a new node and link the next/previous nodes

set the last nodes to have the same pair and make otherNode’s next pointer point to nullptr

set tail equal to last node

* Assignment Operator:

if the object is being set to itself

return a pointer to this

create a new temp map

swap temp map with this

return a pointer to this

* Map::empty function:

if map is empty

return true

* Map::size function:

return number of nodes

* Map::insert function:

iterate through linked list and if key already exists in linked list return false

if list is empty

add a new node and set new node’s m\_map values, new had, tail, next, and previous pointers

else create new node at end of linked list

add new node and set new Node’s m\_map values, tail, next, and previous pointer

increment size

* Map::update function:

iterate through linked list

if key already exists

change the value for that key to the value in the parameter

* Map::insertOrUpdate function:

iterate through linked list

if key already exists in linked list, change the value for that key to that value in the parameter

if list is empty

create new Node and set new Node’s m\_map values, new head, tail ,next, previous pointer

else create new node at end of linked list

create new Node and set new node’s m\_map values, new head, tail, next, previous pointer

increment size

* Map::erase function:

iterate through linked list

if key already exists

if item we want to delete is the first node

delete first node and update head and node pointers

else item we want to delete is a middle or last node

point to node above the node we want to delete

if node we want to delete is last node

delete last node and update node and tail pointers

else node we want to delete is in the middle of linked list

delete node and update node pointers

* Map::contains function:

iterate through linked list

if key already exists in linked list

return true

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

iterate through linked list

if key already exists in linked list

set value to the value in the map which that key maps to

return true

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

iterate through linked list with pointer to Node p

iterate through linked list again with pointer to Node q

if p node is greater than q node

increment count

if i is equal to count

copy key/value pair into the parameters

return true;

* Map::swap function:

swap head pointers

swap tail pointers

swap size

* merge function:

if m1 is empty, result is just m2

if m2 is empty, result is just m1

copy result into m1

iterate through m2

if key is not already in result

add key/value pair into result

else key already exists in result

get value already present in result

if value equals value in key/value pair in map

continue

else value has a different value than the one in result

erase that key/value type altogether

set bool to false

return bool

* reassign function:

copy m into result

if m is just one pair

return

iterate through m

arrange m into alphabetical order and put that into result

iterate through each pair in result

if key/value pair is at front

shift head node’s value to the next node’s value

if pair is in middle

shift current node’s value to the next node’s value

if pair is at end

update head node’s value with the last pair’s value

1. Test Cases

* Map m; // tests default constructor
* assert(m.size() == 0) // tests size
* assert(m.empty()) // tests when map is empty
* assert(m.insert("a", 1)) // tests insert function
* assert(m.size() == 1)) // tests size
* assert(!m.empty()) // tests when map is not empty
* assert(!m.update("b", 2)) // tests update function when key does not exist in map
* assert(m.update("a", 10)) // tests update function when key does exist in map
* assert(!n.update("b", 10)) // tests update function when map is empty
* assert(m.update("b", 30)) // tests update function when updating first node
* assert(m.update("e", 30)) // tests update function when updating last node
* assert(m.insertOrUpdate("b", 2)) // tests insertOrUpdate function when key does not exist in map
* assert(m.insertOrUpdate("b", 20)) // tests insertOrUpdate function when key does exist in map
* assert(!m.erase("c")) // tests erase function when key does not exist in map
* assert(m.insert("d", 40)) // tests insert function when inserting a node into a map already containing nodes
* assert(!m.insert("d", 50)) // tests insert function when key already exists in map
* assert(m.erase("a")) // tests erase function when erasing the node at top
* assert(m.erase("d")) // tests erase function when erasing a node in the middle
* assert(m.erase("f")) // tests erase function when erasing a node at the end
* assert(!m.contains("a")) // tests contains function when key is not in map
* assert(m.contains("b")) // tests contains function when key exists in map
* assert(!n.contains("a")) // tests contains function when map is empty
* assert(m.get("b", o) && o == 20) // tests get function when key exists in map
* assert(!m.get("c", o) && o == 5) // tests get function when key does not exist in map
* assert(!n.get("a", o)) // tests get function when map is empty
* assert(m.get(0, k, v) && k == "b") // tests second get function when i is the first node in map
* assert(m.get(1, k, v) && k == "e") // tests second get function when i is second node(middle) in map
* assert(m.get(2, k, v) && k == "f") // tests second get function when i is the last node in the map
* assert(!m.get(-1, k, v) && k == "xxx") // tests second get function when i is less than 0
* assert(!m.get(3, k, v) && k == "xxx") // tests second get function when i is equal to size
* assert(!n.get(1, k, v) && k == "xxx") // tests second get function when map is empty
* (m.swap(m1)) // tests swap function
* m.swap(n) // tests swap function when one map is empty
* n.swap(q) // tests swap function when both maps are empty
* Map q = m // tests copy constructor
* q = m1 // tests assignment operator
* assert(merge(f, g, result)) // tests merge function when both maps have a same key/value pair
* assert(merge(f, n, result)) // tests merge function when one map is empty
* assert(merge(n, i, result)) // tests merge function when both maps are empty
* assert(!merge(f, g, result)) // tests merge function when both maps have the same key that map to different values
* reassign(i, y) // tests reassign function
* reassign(i, y) // different test for reassign function when there is only one pair in map
* reassign(i, y) // different test for reassign function when there are no pairs in map
* reassign(i, i) // tests reassign function when m1 and result both refer to same map