```cpp
// Map.cpp

#include "Map.h"

#include <iostream>
using namespace std;

Map::Map()
{
   head = nullptr;
   tail = nullptr;
   m_size = 0;
}

Map::Map(const Map& other)
{
   head = nullptr;
   tail = nullptr;
   m_size = 0;

   for (int i = other.size() - 1; i >= 0; i--)
//Traverse down because this Map adds
nodes to the front
   {
      KeyType copyKey;
      ValueType copyValue;
      other.get(i, copyKey, copyValue); //Get
the data from the other map
      insert(copyKey, copyValue); //And
insert them in this map
   }
}

Map& Map::operator=(const Map& rhs)
{
   if (this != &rhs) // Check to see if a Map
is being assigned to itself
   {
      Map temp(rhs); //If not,create a temp
Map to store the data
      swap(temp); //Finally, swap this map
with the temp map (which resembles rhs)
   }
   return *this;
}

Map::~Map()
{
   Node* curr;
   curr = head;
   while (curr != nullptr)
   {
      Node* n = curr->next;
      delete curr;
      curr = n;
      head = n;
   }
   tail = nullptr;
}

bool Map::empty() const
{
   return head == nullptr; //If head equals
nullptr, there are no nodes in the Map
}

int Map::size() const
{
   return m_size;
}

bool Map::insert(const KeyType& key,
const ValueType& value) //Add nodes to the
front of the Map
{
   if (contains(key)) //Check if the key
already exists in the map by calling contains
      return false;

   Node* newNode;
   newNode = new Node; //If it is a new
key, then dynamically allocate a new node
   newNode->m_nodeMap.key = key; //Set
the new node's data to the key and value
   newNode->m_nodeMap.value = value;
   newNode->next = nullptr;
   newNode->prev = nullptr;

   //Two cases: 1. either the map is empty, or
2. the map already has node(s)

   if (empty()) //Check if the map is empty,
if so link up the head and tail to new node
   {
      head = newNode;
      tail = newNode;
   }
   else //If map already has node(s), then add
new node to the front of the Map
   {
      newNode->next = head;
      head->prev = newNode;
```

```
        head = newNode;
    }
    m_size++;
    return true;
}

bool Map::update(const KeyType& key,
const ValueType& value)
{
    if(!contains(key)) // Check to see if the
Map already contains the key
        return false;
    Node* curr = head;
    if (head->m_nodeMap.key == key) //
Check if the key is in the first node of the
Map
        head->m_nodeMap.value = value;
    while (curr != nullptr) // While p is not at
the end of the list
    {
        if (curr->next != nullptr && curr-
>next->m_nodeMap.key == key) //Checking
on next node's data
            curr->next->m_nodeMap.value =
value;
        curr = curr->next; //Traverse to the next
node in the Map
    }
    return true;
}

bool Map::insertOrUpdate(const KeyType&
key, const ValueType& value)
{
    if (!contains(key)) //If the key isn't
already in Map, then insert it
    {
        insert(key, value);
    }
    else
    {
        update(key, value); //If key is already
in Map, reassign its value
    }
    return true;
}

bool Map::erase(const KeyType& key)
{
    if (empty() || contains(key) == false) //If
the map is empty or the key isn't in Map,
return false
        return false;

    if (size() == 1) // If there is only one node
    {
        Node* killMe = head; //Create a
pointer to point to the node to be deleted
        head = nullptr; //Reassign head and tail
to point to no items in the Map
        tail = nullptr;
        delete killMe;
    }
    else if (head->m_nodeMap.key == key) //
If the node is the first one
    {
        Node* killMe = head;
        head = killMe->next;
        head->prev = nullptr;
        delete killMe;
    }
    else if (tail->m_nodeMap.key == key) //
If the node is the last one
    {
        Node* killMe = tail;
        tail = killMe->prev;
        tail->next = nullptr;
        delete killMe;
    }
    else // If the node is somewhere in the
middle of the list
    {
        Node* curr = head; //Create a pointer to
traverse through the Map
        while (curr != nullptr)
        {
            if (curr->next != nullptr && curr-
>next->m_nodeMap.key == key) //Check
next node's data
                break;
            curr = curr->next; //Traverse to next
node in the Map
        }
        Node* killMe = curr->next; //If node is
found, then create a pointer to point to that
node
        Node* n = killMe->next;
        curr->next = killMe->next; //Relink the
Map around the node to be deleted
        n->prev = curr;
        delete killMe;
    }
```

```
    m_size--;
    return true;
}

bool Map::contains(const KeyType& key)
const
{
    if (head == nullptr) //If the map is empty
        return false;
    if (head->m_nodeMap.key == key) //If
there is one node, and it equals key
        return true;
    Node* curr = head;
    while (curr != nullptr) //To validate that p
points to a valid node
    {
        if (curr->next != nullptr && curr-
>next->m_nodeMap.key == key) //Check
next node's data
            return true;
        curr = curr->next; //Traverse to the next
nodes of the list
    }
    return false;
}

bool Map::get(const KeyType& key,
ValueType& value) const
{
    if (contains(key)) //If the key is present in
the Map
    {
        Node* curr = head; //Create a pointer to
traverse through the Map
        while (curr != nullptr)
        {
            if (curr->m_nodeMap.key == key)
//If key is first node in the Map
            {
                value = curr->m_nodeMap.value;
                return true;
            }
            if (curr->next != nullptr && curr-
>next->m_nodeMap.key == key) //Check
next node's data
                break;
            curr = curr->next;
        }
        value = curr->next-
>m_nodeMap.value; //Reassign the given
key's value
        return true;
    }
    return false;
}

bool Map::get(int i, KeyType& key,
ValueType& value) const
{
    if (i < 0 || i >= size())
        return false;

    Node* curr = head;
    for (int j = 0; j < i; j++) //Go one by one,
getting a given key/value pair
        curr = curr->next;

    key = curr->m_nodeMap.key;
    value = curr->m_nodeMap.value;
    return true;
}

void Map::swap(Map& other)
{
    //Swap the sizes

    int tempSize = m_size;
    m_size = other.m_size;
    other.m_size = tempSize;

    //Swap the pointers to point to each
other's Map
    Node* tempHead = head;
    Node* tempTail = tail;

    head = other.head;
    tail = other.tail;

    other.head = tempHead;
    other.tail = tempTail;

}

bool combine(const Map& m1, const Map&
m2, Map& result)
{
    //Create two temporary maps to avoid
aliasing
    Map temp1 = m1;
    Map temp2 = m2;
    Map emptyMap; //Create an empty map
```

```
and clear result by assigning it the empty
map
    result = emptyMap;
    bool wrongMatch = true; //To check if
two maps have the same keys, but different
values

    for (int i = 0; i < temp1.size(); i++)
//Traverse through the first Map
    {
        KeyType k;
        ValueType v;
        temp1.get(i, k, v); //Call get to retrieve
the keys in first Map
        if (! temp2.contains(k)) //If the first
map has the given key, but not the second
map
            result.insert(k, v); //Then add the
key/value pair to result Map
        else //If they both contain the key,
check to see if they
        {
            ValueType v1;
            temp2.get(k, v1); //If they both
contain the key, check to see if they have
same value
            if (v == v1)
                result.insert(k, v); //If so, add the
key/value pair to result Map
            else
                wrongMatch = false; //If different
values, then don't add and return false
        }
    }

    for (int i = 0; i < temp2.size(); i++)
//Traverse through second map to check for
new keys
    {
        KeyType k;
        ValueType v;
        temp2.get(i, k, v);
        if (! temp1.contains(k))
            result.insert(k, v);
    }
    return wrongMatch; //If both maps have
the same key, but different values, then
return false
}

void subtract(const Map& m1, const Map&
m2, Map& result)
{
    Map temp1 = m1; //Create two temporary
maps to avoid aliasing
    Map temp2 = m2;
    Map emptyMap; //Create an empty map,
and clear result map by assigning it the
empty map
    result = emptyMap;

    for (int i = 0; i < temp1.size(); i++)
//Traverse through the first Map
    {
        KeyType k;
        ValueType v;
        temp1.get(i, k, v);
        if (! temp2.contains(k)) //Find keys that
are present in first map, and not in second
map
            result.insert(k, v); //Insert the unique
keys in the first map to the result map
    }
}
```

```
// newMap.cpp

#include "newMap.h"
#include <iostream>
#include <cstdlib>
using namespace std;

Map::Map(int capacity)
: m_size(0),
m_capacity(capacity)
{
    if (capacity < 0)
    {
        cout << "A Map
capacity must not be
negative." << endl;
        exit(1);
    }
    m_data = new
Pair[m_capacity];
}

Map::Map(const Map& other)
: m_size(other.m_size),
m_capacity(other.m_capacity)
{
    m_data = new
Pair[m_capacity];
    for (int k = 0; k <
m_size; k++)
        m_data[k] =
other.m_data[k];
}

Map::~Map()
{
    delete [] m_data;
}

Map& Map::operator=(const
Map& rhs)
{
    if (this != &rhs)
    {
        Map temp(rhs);
        swap(temp);
    }
    return *this;
}

bool Map::erase(const
KeyType& key)
{
    int pos = find(key);
    if (pos == -1)  // not
found
        return false;

    // Move last array item
to replace the one to be
erased

    m_size--;
    m_data[pos] =
m_data[m_size];
    return true;
}

bool Map::get(const KeyType&
key, ValueType& value) const
{
    int pos = find(key);
    if (pos == -1)  // not
found
        return false;
    value =
m_data[pos].m_value;
    return true;
}

bool Map::get(int i, KeyType&
key, ValueType& value) const
{
    if (i < 0  ||  i >=
m_size)
        return false;
    key = m_data[i].m_key;
    value =
m_data[i].m_value;
    return true;
}

void Map::swap(Map& other)
{
    // Swap the m_data
pointers to dynamic arrays.

    Pair* tempData = m_data;
    m_data = other.m_data;
    other.m_data = tempData;
```

```
    // Swap sizes

    int t = m_size;
    m_size = other.m_size;
    other.m_size = t;

    // Swap capacities.

    t = m_capacity;
    m_capacity =
other.m_capacity;
    other.m_capacity = t;
}

int Map::find(const KeyType&
key) const
{
    // Do a linear search
through the array.

    for (int pos = 0; pos <
m_size; pos++)
        if (m_data[pos].m_key
== key)
            return pos;
    return -1;
}

bool
Map::doInsertOrUpdate(const
KeyType& key, const
ValueType& value,

bool mayInsert, bool
mayUpdate)
{
    int pos = find(key);
    if (pos != -1)  // found
    {
        if (mayUpdate)

m_data[pos].m_value = value;
        return mayUpdate;
    }
    if (!mayInsert)  // not
found, and not allowed to
insert
        return false;
    if (m_size == m_capacity)
// no room to insert
        return false;
    m_data[m_size].m_key =
key;
    m_data[m_size].m_value =
value;
    m_size++;
    return true;
}
```