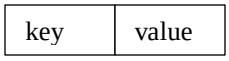
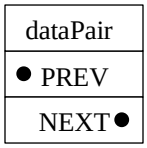
Nathan Tung
Com Sci 32 (Lec 2)
January 30, 2012

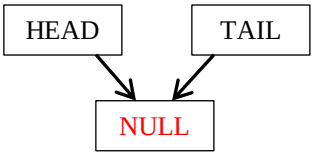
CS31 Project 2 Report

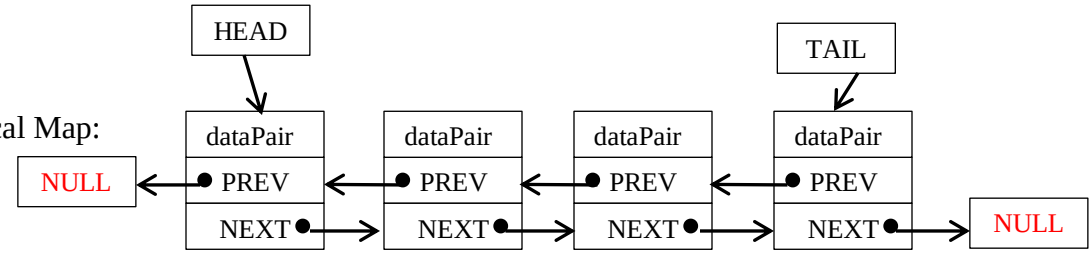
a. My design of the Map double-linked list comprises mainly of two structures, mapNode and dataPair. dataPair holds two values: a key (of KeyType) that is essentially connected to a value (of ValueType). On the other hand, each mapNode contains a dataPair and two pointers linking that mapNode to other mapNodes preceding and succeeding it.

The Map itself has a head pointer for the first mapNode in the list, and a tail pointer for the last. When new keys/values are inserted, a new mapNode is attached to the end by the tail, and that node becomes the new tail. Nodes are basically arranged from oldest-added items from the left to the most recently-added items on the right. The list I implemented is not circular, nor does it use a dummy node.

dataPair: 

mapNode: 

Empty Map: 

Typical Map: 

b. **Map()**

Initialize Map size to 0
Set head and tail pointers to NULL

~Map()

If Map is not empty
    If head and tail point to the same thing, delete the mapNode
    Else, create a "temp" pointer to head's mapNode
        While temp is not NULL:
            Set temp to the next mapNode
            Delete the mapNode right before temp
        Delete the tail (which isn't deleted since temp would be NULL)

Nathan Tung
Com Sci 32 (Lec 2)
January 30, 2012

Map(const Map& other)

Initialize Map size to 0
Set head and tail pointers to NULL
For each mapNode in other:
    Store the key and value of that mapNode into temporary values k, v
    Insert k, v into the current Map

Map& operator=(const Map& rhs)

If the current Map and rhs are not equal
    Copy-construct a "temp" Map using rhs
    Swap current Map with temp
Return a pointer to current Map

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

If Map already contains key, return false
Otherwise, create a new "newNode" mapNode
    Assign newNode the key and value in parameters
    Set its prev/next pointers to NULL
If Map is currently empty
    Make head and tail point to newNode
Else, set newNode's prev to tail and set tail's next to newNode
    Then make tail point to newNode
Increment counter for Map size and return true

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

If Map doesn't contain key, return false
Otherwise, create a "temp" mapNode pointer to head
While the key of the dataPair held in temp's mapNode doesn't match parameter key:
    Set temp to temp's next node
Set the value of the dataPair held in temp's mapNode to parameter value and return true

bool insertOrUpdate(const KeyType& key, const ValueType& value)

If Map already contains key, update the key
Else, insert the key
Return true

bool erase(const KeyType& key)

If Map doesn't contain key, return false
Otherwise, if there is only one mapNode in Map
    Delete it, set head/tail to point to NULL, and decrement Map size counter

Otherwise, create a "temp" mapNode pointer to head
While the pair-key of the mapNode temp points to doesn't match the parameter key:
    Set temp to its next mapNode
If temp is the head
    Set head to next pointer after temp and set head's prev to NULL and delete temp
    Delete temp
Else, if temp is tail
    Set tail to prev pointer before temp and set tail's next to NULL and delete temp
Else, make two pointers "before" and "after" for temp's prev/next mapNodes
    Set before's next to after, then set after's prev to before and delete temp
Decrement Map's size counter and return true

bool contains(const KeyType& key) const

Make a "temp" pointer to head's mapNode
If head is NULL, return false
Otherwise, while temp is not NULL:
    If the key in the dataPair of temp's mapNode matches the parameter key
        Return true
    Otherwise, set temp to temp's next mapNode
Return false (since no matches found)

bool get(const KeyType& key, ValueType& value) const

If Map doesn't contain key, return false
Otherwise, make a "temp" pointer to head's mapNode
While the key in the dataPair of temp's mapNode doesn't match the parameter key:
    Set temp to its next mapNode
Set parameter value to the value in the dataPair of temp's mapNode
Return true

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

If integer i is less than 0 or at least as big as Map's size, return false
Otherwise, make a "temp" pointer to head's mapNode
For integer-i number of loops:
    Set temp to its next mapNode
Set parameter key and value to key and value in the dataPair of temp's mapNode
Return true

void swap(Map& other)

Make "tempHead" and "tempTail" mapNode pointers
    Set them to head and tail, respectively

Set current Map's head and tail to other's head and tail
Set other Map's head and tail to tempHead and tempTail
Using a temporary "currentSize" integer, switch size counter for current Map and other

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

Create Maps "m1Temp" and "m2Temp"
    Set them equal to m1 and m2, respectively
Initialize a boolean "noMismatches" to true
While result Map is not empty:
    Get key and value of first mapNode and delete it (until result Map is empty)
For integer-i from 0 to m1Temp's size:
    Get key and value from corresponding mapNode
    If m2Temp doesn't contain that key, insert it into result
    Otherwise, get value from the same-key mapNode in m2Temp
        If both values are equal, insert it into result
        Else, set boolean noMismatches to false
For integer-i from 0 to m2Temp's size:
    Get key and value from corresponding mapNode
    If m1Temp doesn't contain that key, insert it into result
Return boolean noMismatches

void subtract(const Map& m1, const Map& m2, Map& result)

Create Maps "m1Temp" and "m2Temp"
    Set them equal to m1 and m2, respectively
While result Map is not empty:
    Get key and value of first mapNode and delete it (until result Map is empty)
For integer-i from 0 to m1Temp's size:
    Get key and value from corresponding mapNode
    If m2Temp doesn't contain that key, insert it into result

c. My exhaustive test cases dedicate many Maps to the more complicated functions (such as insert/update, erase, combine/subtract, swap, and the constructor/equals operand. Other functions are still tested (such as contains, get, empty, size, etc.) throughout the code and often used to double-check the results of other functions. The test cases are as follows:

Nathan Tung
Com Sci 32 (Lec 2)
January 30, 2012

```
KeyType k;
ValueType v;

Map hello; //Initiate a new map without specification
assert(hello.empty()); //Check that hello has no nodes
assert(hello.size()==0); //Same as above (test multiple functions)
assert(hello.erase("joke")==false); //Check that there's nothing to delete
assert(hello.contains("troll")==false); //Check that there's nothing contained

assert(hello.insert("Head", 0));
assert(!hello.empty()); //Check that hello is no longer empty
assert(hello.size()==1); //Check that the single node has been inserted
assert(hello.contains("Head")); //Check that inserted nodes are contained
assert(!hello.insert("Head", 1)); //Check that duplicate keys cannot be inserted
assert(hello.size()==1); //Check that the size has not changed

hello.get("Head", v);
assert(v==0); //Check that the node with key "Head" is holding the right value

assert(!hello.erase("blah")); //Check that a node that does not exist cannot be erased
assert(hello.update("Head", 9)); //Check that the node with key "Head" is found and
updated

hello.get("Head", v);
assert(v==9); //Check that that node is holding the updated value

assert(hello.insert("Between", 8)); //Check that another node can be inserted

hello.get("Between", v);
assert(v==8); //Check that that node is holding the right value

hello.insertOrUpdate("Tail", 11);
hello.insertOrUpdate("Between", 10);

assert(hello.size()==3); //Check that the right number of nodes exists

for(int i=0; i<hello.size(); i++)
{
	hello.get(i, k, v);
	cerr << k << "->" << v << endl;    //Should print out keys and values such that
Head->9; Between->10; Tail->11
}

assert(hello.contains("Head") && hello.contains("Between") && hello.contains("Tail"));
//Make sure the right nodes are contained
assert(!hello.contains("head")); //Check that other nodes are not considered contained

cout << "Passed basic tests involving one map." << endl;

//Testing copy constructor and equals operand

Map helloCopy1(hello); //test that the copy constructor makes an exact copy of the map

assert(helloCopy1.size()==3); //Check that the right number of nodes exists

for(int i=0; i<helloCopy1.size(); i++)
{
```

```
	helloCopy1.get(i, k, v);
	cerr << k << "->" << v << endl;	//Should print out keys and values such that
Head->9; Between->10; Tail->11
}

Map helloCopy2=hello; //test that the copy constructor makes an exact copy of the map
assert(helloCopy2.size()==3); //Check that the right number of nodes exists

for(int i=0; i<helloCopy2.size(); i++)
{
	helloCopy2.get(i, k, v);
	cerr << k << "->" << v << endl;	//Should print out keys and values such that
Head->9; Between->10; Tail->11
}

while(!helloCopy2.empty())
{
	helloCopy2.get(0, k, v);
	helloCopy2.erase(k); //Check that the map can be emptied by erase
	cerr << k << "->" << v << " removed from helloCopy2" << endl;
}

Map blank; //Create a blank map
helloCopy1=blank;
assert(helloCopy1.empty()); //Check that equals operand works for empty maps

Map anotherBlank(blank);
assert(anotherBlank.empty()); //Chcek that copy constructor works for empty maps
```

cout << "Passed basic tests involving copy constructor and equals operand." << endl;

```
Map oneNode;
oneNode.insert("blah", 1);
Map copyOneNode(oneNode); //Test the copy constructor with only one node
assert(copyOneNode.size()==1);  //Check that the size is correct
assert(copyOneNode.contains("blah")); //Check that the key is contained in the new map

Map goodbye;
hello.swap(goodbye); //Swap hello with an empty map
assert(hello.empty()); //Make sure the amount of elements has switched
assert(goodbye.size()==3);

for(int i=0; i<goodbye.size(); i++)
{
	goodbye.get(i, k, v);
	cerr << k << "->" << v << endl;
}

goodbye.swap(hello); //Swap hello and goodbye back to normal
assert(goodbye.empty());
assert(hello.size()==3);

for(int i=0; i<hello.size(); i++)
{
	hello.get(i, k, v);
```

```
	cerr << k << "->" << v << endl; //Verify that hello and the original goodbye are
the same
}
```

cout << "Passed swap tests." << endl;

```
assert(hello.erase("Between")); //Check that the middle node is removed and size is
updated
assert(hello.size()==2);

assert(hello.erase("Tail")); //Check that the tail node can be removed
assert(hello.size()==1);
assert(hello.erase("Head"));
assert(hello.empty()); //Check that the head node can be removed (when it's the last one)
assert(hello.insert("new head", 5));
assert(hello.insert("new tail", 6));
assert(hello.erase("new head")); //Check that the head node can be removed (when it's NOT
the last node)
assert(hello.size()==1);
```

cout << "Passed erase tests." << endl;

```
Map firstMap;
firstMap.insert("one", 1); //Should be included in combine
firstMap.insert("two", 2); //Should be included in combine
firstMap.insert("three", 3); //Should be included in combine, since the values in
first/secondMap equal
firstMap.insert("four", 4); //Should NOT be included in combine, since the values in
first/secondMap differ

Map secondMap;
secondMap.insert("three", 3);
secondMap.insert("four", 10); //Inconsistent value! Should not be in combined Map
secondMap.insert("five", 5);

Map answer;

assert(combine(firstMap, secondMap, answer)==false);
assert(answer.size()==4); //Check that the right number of nodes exist
for(int i=0; i<answer.size(); i++)
{
	answer.get(i, k, v);
	cerr << k << "->" << v << endl;
}

assert(combine(firstMap, secondMap, secondMap)==false); //Check that combine works in
face of aliasing
assert(secondMap.size()==4);
for(int i=0; i<secondMap.size(); i++)
{
	secondMap.get(i, k, v);
	cerr << k << "->" << v << endl;
}

Map simpleOne;
simpleOne.insert("blah", 1);
```

```
Map simpleTwo;
simpleTwo.insert("bleh", 2);
Map simpleThree;

assert(combine(simpleOne, simpleTwo, simpleThree)); //Check that combine returns true for
unique keys
assert(simpleThree.size()==2);
assert(simpleThree.contains("blah") && simpleThree.contains("bleh")); //Check for the
correct pairs

Map random;
assert(random.insert("eek", 0) && random.insert("aah", 1));
assert(random.size()==2);
Map blankMap;
Map blankResult;

combine(random, blankMap, blankResult); //test subtract with an empty Map as m2
assert(blankResult.size()==2 && blankResult.contains("eek") &&
blankResult.contains("aah"));

combine(blankMap, random, blankResult); //test subtract with an empty Map as m1
assert(blankResult.size()==2 && blankResult.contains("eek") &&
blankResult.contains("aah"));
```

cout << "Passed combine tests." << endl;

```
Map firstMapCopy;
firstMapCopy.insert("one", 1); //Should be included in combine
firstMapCopy.insert("two", 2); //Should be included in combine
firstMapCopy.insert("three", 3); //Should be included in combine, since the values in
first/secondMap equal
firstMapCopy.insert("four", 4); //Should NOT be included in combine, since the values in
first/secondMap differ

Map secondMapCopy;
secondMapCopy.insert("three", 3);
secondMapCopy.insert("four", 10); //Inconsistent value! Should not be in combined Map
secondMapCopy.insert("five", 5);

Map anotherResult;

subtract(firstMapCopy, secondMapCopy, anotherResult);
assert(anotherResult.size()==2);
assert(anotherResult.contains("one") && anotherResult.contains("two")); //Check for the
correct pairs

for(int i=0; i<anotherResult.size(); i++)
{
	anotherResult.get(i, k, v);
	cerr << k << "->" << v << endl;
}

subtract(firstMapCopy, secondMapCopy, firstMapCopy); //Check that subtract works in the
face of aliasing
assert(firstMapCopy.size()==2);
assert(firstMapCopy.contains("one") && firstMapCopy.contains("two")); //Check for the
correct pairs
```

Nathan Tung
Com Sci 32 (Lec 2)
January 30, 2012

```
for(int i=0; i<firstMapCopy.size(); i++)
{
	firstMapCopy.get(i, k, v);
	cerr << k << "->" << v << endl;
}

Map random2;
assert(random2.insert("eek", 0) && random2.insert("aah", 1));
assert(random2.size()==2);
Map blankMap2;
Map blankResult2;

subtract(random2, blankMap2, blankResult2); //test subtract with an empty Map as m2
assert(blankResult2.size()==2 && blankResult2.contains("eek") &&
blankResult2.contains("aah"));

subtract(blankMap2, random2, blankResult2); //test subtract with an empty Map as m1
assert(blankResult2.empty());
```

cout << "Passed subtract tests." << endl;

```
cout << "Passed all of my tests." << endl;
```