Linked List Implementation, Reading

Because of the not-in-use "holes" and the fact that we don't use direct random access of the data array, associative array objects lend themselves very well to data storage using singly-linked lists. There's no capacity issue with linked lists, no holes, and they have for-loops for traversal.

We already developed a node structure for the linked stack and queue in an earlier module. We can borrow from those -- all we have to do is add an attribute for "key". To compare and contrast with the stack template:

```
template <typename V>
class Stack
{
  struct Node
  {
    V value;
    Node* next;
  };

  Node* firstNode;
  int siz;

  public:
  ...
};
```

vs.

```
template <typename K, typename V>
class AssociativeArray
{
  struct Node
  {
    K key;
    V value;
    Node* next;
  };

  Node* firstNode;
  int siz;

  public:
  ...
};
```

Compared to the arrayed associative array, the Node has no "inUse" attribute, since there will be no unused Nodes. Nodes are to be removed and deallocated when key is deleted.

The **specification** is the same no matter what is the implementation -- arrayed or linked or whatever. But the coding of the template member functions will of course change, as will the dynamic memory management functions.

Main Constructor, size, And clear Member Functions

These are identical to the linked stack versions. And as in the linked stack, there is no defaulted parameter to specify initial capacity, because capacity is not a concept associated with our linked implementations. And there's no need for a tail pointer.

The Square Bracket Operator Getter

There is no stack or queue version of this function to call upon as a starting point, so we'll have to compare and contrast with what we *do* have -- the arrayed square bracket operator.

The arrayed square bracket operator traverses the array, looking first for in-use Nodes, and then looking for a match. But the for-loops for traversal are different:

arrayed

```
for (int i = 0; i < cap; i++)
  if (data[i].inUse && data[i].key == key)
    return data[i].value;
```

vs.

linked

```
for (Node* p = firstNode; p; p = p->next)
  if (p->key == key)
    return p->value;
```

If no match is found, the for-loop ends without having exited the function with a return statement. So after the loop and before the function ends, we

create a dummy of **typename V** and return that. Remember -- it's the *getter*.

The Square Bracket Operator Setter

The linked setter works very much like the getter version, and not so much like the arrayed version -- because the latter has "inUse" tracking and capacity doubling. So it starts with the exact same for-loop as the getter. If it returns out of that loop, fine.

But if the loop ends and no match was found, then (1) create a new node, (2) set its key to the parameter key (to which no match was found), (3) set its value to the default for its typename, and (4) insert it at the head (because it's easiest, and why not?). Don't forget to increment size at some point, and finally return the head node's value.

```
++siz;
Node* temp = new Node;
temp->key = key;
temp->value = V( );
temp->next = firstNode;
firstNode = temp;
return firstNode->value;
```

containsKey Getter And deleteKey Setter

The getter starts with the same for-loop that the square bracket operators have, but it returns "true" instead of a value. If the loop completes without having found a match, return "false".

The setter is a bit more involved, because it means removing a Node from the *middle* or *either end* of a linked list. For that, we need to recall what we learned in Comsc-165 about how to remove a node from a linked list -- hopefully we all remember that, or can find it in our notes. It should look like this (for nodes named "Student" with a key named "name"):

```
// FROM COMSC-165 LINKED LISTS:
// find the node for the student Joe
Student* p, *prev; // declare above loop so that it survives below the loop
for (p = firstStudent, prev = 0; p; prev = p, p = p->next)
  if (p->name == "Joe")
    break;

// if found (that is, p did not run off the end of the list)
if (p)
{
  if (prev) prev->next = p->next; // skips over the node at p
  else firstStudent = p->next; // there's a new head, possibly zero
  delete p; // we're done with this node
}
```

Adapting this to our use:

```
// find the matching node
Node* p, *prev; // declare above loop so that it survives below the loop
for (p = firstNode, prev = 0; p; prev = p, p = p->next)
  if (p->key == key)
    break;

// if found (that is, p did not run off the end of the list)
```

```
// if found (that is, p did not run off the end of the list)
if (p)
{
  --siz;
  if (prev) prev->next = p->next; // skips over the node at p
  else firstNode = p->next; // there's a new head, possibly zero
  delete p; // we're done with this node
}
```

Just be sure to decrement the size tracker.

The keys Getter

Again, the same traversing for-loop. But there's no in-use check -- just push the **p->key**'s into the queue of **typename K** and return that C++ STL queue at the end of the function.

Dynamic Memory Management Functions

Since the internal data structure is identical to that of a linked stack, these functions are *identical* to the stack's. But remember to copy the *key* as well as the *value*.