Hash Tables
Separate Chaining

Justification

Hash tables with open addressing are simple to use, but have the problem of filling up and needing to be rehashed into a new larger array to keep performance O(1). Adding separate chaining to a hash table allows it to contain a multiple of the number of items without seriously degrading performance, as long as the hash function does not cause excessive clumping of values.

It is a little more complex, but not too bad. Which one to use depends on the situation. If it is known that the number of items is going to remain fairly small, such as the symbols in a computer program then the best solution is open addressing with a reasonably sized array. For situations where there are a larger number of items or the hash table will be adding items over a longer period of time, then separate chaining is probably worth the effort.

Separate Chaining

The concept behind separate chaining is to have a linked list of values stored at each location instead of storing a single value. The performance is then O(1) for the original hash function and O(N) for searching each linked list where N is the number of items in that list. As long as the average length of the linked lists is small, the cost of this linear search remains small enough that the overall performance can be considered O(1). For example, if there are fewer than three times as many items stores as the table length, then the average search of a linked list is only three steps. In this case, the searching of any one list would be on average O(3), which is considered O(1) since multiplying the time by a constant does not change the O value.

The algorithm is to start with the array initialized to whatever empty pointer is used in the programming language (nullptr, None, null). Then hash the value to get an index and use an addHead at that location. For find, hash the value then use a linked list find at that location. Finally, for delete, hash the value and use a deleteItem method at that location.

There is no need to detect collisions, no need to step the index, and no need to have a special variable for DELETED.

The following graphic shows how such a hash table might logically be arranged. The values were arbitrarily placed, there is no specific hash function that would place unicorn, goat, and cow to the same index. This is shown as a 2D array where each row is a linked list in the hash table.

nullptr			
nullptr			
cat	nullptr		
nullptr			
dog	pig	nullptr	

horse	nullptr		
nullptr			
unicorn	goat	cow	nullptr
nullptr			

Implementation

There are two ways of coding this. One is to have an implicit linked list where each location starts out with a nullptr/None/null and the code for add, find, and remove implement the appropriate linked list algorithm. This is the first version that is shown.

Implicit Linked List

Here are pseudocode methods for addValue, findValue, and removeValue. Note that this particular example creates a new node and adds it at the correct location. This code can be compared with that of a single linked list to show that it is identical.

This assumes that there is a class ChainItem that is a link style class containing a value and a next pointer. This code uses nullptr. Obviously for different languages None or null would be used.

```
// calculate an index from the value
hash(value)
    return value % arraySize


// add a new item in the list with key as value
addValue(value)

    // create a new item with that value
    temp = new ChainItem(value)

    // get the index into the hash array
    index = hash(value)

    // add to head of list at that location
    temp.setNext(theTable[index])
    theTable[index] = temp


// look for an item containing key
findValue(value)
```

```
    // get the index into the hash array
    index = hash(value)

    found = false
    ptr = theTable[index]

    // there is a linked list, so search the list for value
    // if location is empty, loop terminates immediately
    while ptr != nullptr and !found

        if (ptr.getValue() == value)
            found = true
        else
            ptr = ptr.getNext()

    // end of list or found item
    return found


// remove the item containing value
removeValue(value)

    // get the index into the hash array
    index = hash(value)

    // only does something if that location is not empty
    if theTable[index] != nullptr

        // there is a linked list, so search the list for item
        ptr = theTable[index]

        // special case first item
        if ptr.getValue() == value

            // update list, delete item (C++ only)
```

```
            theTable[index] = ptr.getNext()
            delete ptr

        // not head, so walk down rest of list
        // single linked, so have to look ahead
        else
            done = false

            while ptr.getNext() != nullptr and !done

                // is the next one what we want?
                if ptr.getNext().getValue() == value

                        // if so, update list and delete (C++ only)
                        ptr.setNext(ptr.getNext().getNext())
                        delete ptr.getNext()
                        done = true

                else
                        ptr = ptr.getNext()
```

Explicit Linked List

The other option is to create the hash table as an array of linked lists. Then the methods are simpler. For practice in linked list methods, the exercises here require the use of the implicit code above.

Here is an example of how it could be implemented in C++. Other languages will be similar.

```
std::list<int> theTable[SIZE];

// add a new item in the list with key as value
void addValue(int value)
{
   theTable[hash(value)].push_front(value);
}

// look for an item containing key
bool findValue(int value)
{
```

```
    std::list<int>::iterator it;

    it = std::find(theTable[hash(value)].begin(),
                   theTable[hash(value)].end(), value);

    return it != theTable[hash(value)].end();
}

// remove the item containing value
void removeValue(int value)
{
    theTable[hash(value)].remove(value);
}
```