Homework #5

a) **(5 points)** Modify the chained hash table implementation from the lecture notes so that it auto-grows when its load factor exceeds a given value. Auto-growing means increasing the number of buckets in the hash table.

Begin by modifying the constructor to take two additional parameters:

- `maxLoadFactor`: Indicates the maximum load factor the hash table is allowed before it auto-resizes. For example, if this value is 1.5 the has table should auto-grow when the load factor goes above 1.5.
- `resizeMultiplier`: The amount by which the number of buckets is multiplied when resizing the table. For example, if this value is 2 the number of buckets will be doubled each time the has table grows.

The remainder of the hash table interface (public method signatures) should remain unchanged.

Next update the internal implementation of the hash table as follows:

1. Resize the hash table when the load factor exceeds the maximum load factor. The new size of the hash table should be the old size times the `resizeMultiplier`. All elements currently in the hash table must be rehashed and placed into new buckets.
2. Change the method by which hash codes are mapped to buckets to use the multiplication method instead of the division method.

Modify any other parts of the hash table as needed. Note that removing items from the hash table should not cause the number of buckets in the table to shrink (the table only grows, never shrinks).

b) **(3 points)** Implement a `main` method in a class named `Homework5` that demonstrates inserts and lookups with the auto-resizing hash table. This method should initialize the hash table to a small number of buckets then begin inserting integers until a resize occurs. After each insert the program should output the following information:

- Number of buckets in the table
- Number of elements in the table
- The table's load factor
- The table's max load factor
- The table's resize multiplier

For example, if the hash table was initialized to start with 5 buckets, a max load factor of 0.5, and a resize multiplier of 2.0, the following output should be displayed as elements are inserted:

```
buckets 5, elements 1, lf 0.20, max lf 0.5, resize multiplier 2.0
buckets 5, elements 2, lf 0.40, max lf 0.5, resize multiplier 2.0
buckets 10, elements 3, lf 0.33, max lf 0.5, resize multiplier 2.0
```

Note that in the 3rd line of output the number of buckets has doubled. This happened because the load factor that would result from inserting the 3rd element would have caused the load factor to exceed the max load factor (0.5) so the hash table was auto-resized.

After the resize occurs, your program must demonstrate successfully looking up a value that was inserted before the resize and must also demonstrate unsuccessfully looking up a value that does not exist in the table.

c) **(1 point)** Answer the following questions:

1. What is the Big-O execution performance of an insert now that auto-resizing can take place?
2. Why do you think you were required to change the hash table to use the multiplication method instead of the division method to map hash codes to buckets?

d) **(1 point)** Make sure your source code is well-commented, consistently formatted, uses no magic numbers/values, and follows programming best-practices.

Turn in all source code (including your modified hash table), program output, diagrams, and answers to questions in a single PDF document.