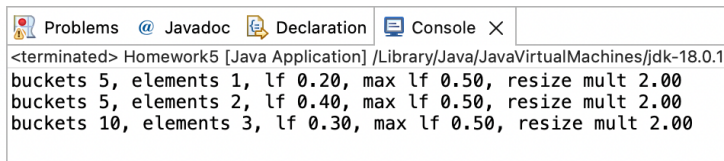


## Homework 5

### Question c)

1. The Big-O of an insertion now is  $O(n)$ , because it requires rehashing all elements from old hash table into new buckets when it is resizing.
2. We're required to change to use multiplication method instead of the division method, because the number of buckets would no longer be a prime every time the number of buckets become A times when resizing required ( $A = \text{resizeMultiplier}$ ). So we change it to the multiplication method, which doesn't require the number of buckets to be a prime.

### Output:



```
<terminated> Homework5 [Java Application] /Library/Java/JavaVirtualMachines/jdk-18.0.1
buckets 5, elements 1, lf 0.20, max lf 0.50, resize mult 2.00
buckets 5, elements 2, lf 0.40, max lf 0.50, resize mult 2.00
buckets 10, elements 3, lf 0.30, max lf 0.50, resize mult 2.00
```

### Source code:

```
// Homework5.java

package com;

public class Homework5 {
    public static void main(String[] args) {
        // Create a new hashtable
        ChainedHashTable<Integer, Double> testTable = new ChainedHashTable<> (5, 0.5,
2);

        // Insert three elements
        testInsert(1, 5.0, testTable);
        testInsert(2, 6.0, testTable);
        testInsert(3, 7.0, testTable);
    }

    // Method to insert and display the result
    private static <K, V> void testInsert(K key, V value, ChainedHashTable<K, V>
testTable) {
        testTable.insert(key, value);
        System.out.printf(
            "buckets %d, elements %d, lf %.2f, max lf %.2f, resize
mult %.2f\n",
                testTable.getLength() ,
                key,
                (double)testTable.getSize()/testTable.getLength() ,
                testTable.getMaxLoadFactor(),
                testTable.getResizeMultiplier()
            );
    }
}
```

```

//ChainedHashTable.java

package com;

import java.util.Iterator;
import java.util.NoSuchElementException;

public class ChainedHashTable<K, V> {
    // Table of buckets
    private SinglyLinkedList<KeyValuePair<K, V>>[] table;

    private int size;
    private double maxLoadFactor;
    private double resizeMultiplier;

    public ChainedHashTable() {
        this(997, 1.5, 2); // A prime number of buckets
    }

    @SuppressWarnings("unchecked")
    public ChainedHashTable(int buckets, double maxLoadFactor, double resizeMultiplier) {
        this.maxLoadFactor = maxLoadFactor;
        this.resizeMultiplier = resizeMultiplier;
        // Create table of empty buckets
        table = new SinglyLinkedList[buckets];
        for (int i = 0; i < table.length; ++i) {
            table[i] = new SinglyLinkedList<KeyValuePair<K, V>>();
        }

        size = 0;
    }

    public int getSize() {
        return size;
    }

    public boolean isEmpty() {
        return getSize() == 0;
    }

    public int getLength() {
        return table.length;
    }

    public double getMaxLoadFactor() {
        return maxLoadFactor;
    }

    public double getResizeMultiplier() {
        return resizeMultiplier;
    }

    @SuppressWarnings("unchecked")
    public void insert(K key, V value) throws
        IllegalArgumentException,
        DuplicateKeyException {
        if (key == null) {
            throw new IllegalArgumentException("key must not be null");
        }
        if (contains(key)) {
            throw new DuplicateKeyException();
        }
        // if it doesn't exceed the max load factor
        if ((double)(size+1)/table.length <= maxLoadFactor) {
            getBucket(key).insertHead(new KeyValuePair<K, V>(key, value));
            ++size;
        }
        // if it exceeds the max load factor
        else {
            // keep the old table
            SinglyLinkedList<KeyValuePair<K, V>>[] cpyTable = table;
            // Create a new table
            table = new SinglyLinkedList [(int)resizeMultiplier*table.length];
            for (int i=0; i<table.length; i++) {
                table[i] = new SinglyLinkedList<KeyValuePair<K, V>>();
            }
            // Copy from old hashtable
            for(int i =0; i < cpyTable.length; i++) {
                for(SinglyLinkedList<KeyValuePair<K, V>>.Element elem =

```

```

        cpyTable[i].getHead();
        elem != null;
        elem = elem.getNext() {
            table[i].insertHead(new KeyValuePair<K, V>(elem.getData().getKey(),
elem.getData().getValue()));
        }
        // insert new element
        getBucket(key).insertHead(new KeyValuePair<K, V>(key, value));
        ++ size;
    }
}

public V remove(K key) throws
    IllegalArgumentException,
    NoSuchElementException {
    if (key == null) {
        throw new IllegalArgumentException("key must not be null");
    }

    // If empty bucket
    SinglyLinkedList<KeyValuePair<K, V>> bucket = getBucket(key);
    if (bucket.isEmpty()) {
        throw new NoSuchElementException();
    }

    // If at head of bucket
    SinglyLinkedList<KeyValuePair<K, V>>.Element elem = bucket.getHead();
    if (key.equals(elem.getData().getKey())) {
        --size;
        return bucket.removeHead().getValue();
    }

    // Scan rest of bucket
    SinglyLinkedList<KeyValuePair<K, V>>.Element prev = elem;
    elem = elem.getNext();
    while (elem != null) {
        if (key.equals(elem.getData().getKey())) {
            --size;
            return bucket.removeAfter(prev).getValue();
        }
        prev = elem;
        elem = elem.getNext();
    }

    throw new NoSuchElementException();
}

public V lookup(K key) throws
    IllegalArgumentException,
    NoSuchElementException {
    if (key == null) {
        throw new IllegalArgumentException("key must not be null");
    }

    // Scan bucket for key
    SinglyLinkedList<KeyValuePair<K, V>>.Element elem =
        getBucket(key).getHead();
    while (elem != null) {
        if (key.equals(elem.getData().getKey())) {
            return elem.getData().getValue();
        }
        elem = elem.getNext();
    }

    throw new NoSuchElementException();
}

public boolean contains(K key) {
    try {
        lookup(key);
    } catch (IllegalArgumentException ex) {
        return false;
    } catch (NoSuchElementException ex) {
        return false;
    }

    return true;
}

```

```

}

private SinglyLinkedList<KeyValuePair<K, V>> getBucket(K key) {
    // Multiplication Method
    long A = (long) ((Math.sqrt(5)-1)/2);
    return table[(int) Math.floor(table.length*Math.floorMod(A*key.hashCode(), 1))];
}

private class KeysIterator implements Iterator<K> {
    private int remaining; // Number of keys remaining to iterate
    private int bucket; // Bucket we're iterating
    private SinglyLinkedList<KeyValuePair<K, V>>.Element elem;
    // Position in bucket we're iterating

    public KeysIterator() {
        remaining = ChainedHashTable.this.size;
        bucket = 0;
        elem = ChainedHashTable.this.table[bucket].getHead();
    }

    public boolean hasNext() {
        return remaining > 0;
    }

    public K next() {
        if (hasNext()) {
            // If we've hit end of bucket, move to next non-empty bucket
            while (elem == null) {
                elem = ChainedHashTable.this.table[++bucket].getHead();
            }

            // Get key
            K key = elem.getData().getKey();

            // Move to next element and decrement entries remaining
            elem = elem.getNext();
            --remaining;

            return key;
        } else {
            throw new NoSuchElementException();
        }
    }
}

public Iterable<K> keys() {
    return new Iterable<K>() {
        public Iterator<K> iterator() {
            return new KeysIterator();
        }
    };
}
}

```