Data Structures and Algorithms in Java (CSE-41321)

UCSD Extension - Summer 2021 (157162)

Assignment #5

Java Implementation of a Hash Table with Separate Chaining

Background

In this exercise I modified the Java source code of a hash table implementation. This implementation uses "separate chaining" as its anti-collision strategy. Collisions occur when the program's hash function computes the same slot number for multiple keys. My assignment was to add an automatic resizing method, so that the underlying array of linked lists would automatically increase in size when the ratio of the number of elements to the number of "buckets" exceeded a predetermined threshold. Furthermore, I changed the formula used by the hash method from division to multiplication.

Solution

The source code that I appropriated, modified or produced myself is contained in a single file named *Homework5. java*. The classes it contains are as follows:

- SinglyLinkedList
  - KeyValuePair
- ChainedHashTable
  - KeysIterator
- DuplicateKeyException
- Driver

In addition, there is a test class, *Homework5Test*, containing the unit tests. This class is based on the TestNG framework, a framework that seems to work more seamlessly than the other one I've used.

The *SinglyLinkedList*, *ChainedHashTable* and *DuplicateKeyException* (and their corresponding inner classes) comprise this implementation of a hash table with separate chaining. The *Driver* class is where the main method resides. It executes a number of statements which demonstrate various functions of the *ChainedHashTable* class.

The Code

Homework5

```
package cse41321.containers;

import org.testng.annotations.Test;

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

public class Homework5 {

    /**
     * This class is intended for use with a hash table.
     *
     * @param <K>
     * @param <V>
     */
    static class KeyValuePair<K, V> {
        private K key;
        private V value;

        public KeyValuePair(K key, V value) {
            this.key = key;
            this.value = value;
        }

        public K getKey() {
            return key;
        }

        public V getValue() {
            return value;
        }
    }

    /**
     * This class implements a singly-linked list which will be used by the ChainedHashTable class.
     *
     * @param <E>
     */
    static class SinglyLinkedList<E> {
        // An element in a linked list
        class Element {
            private E data;
            private Element next;

            // Only allow SinglyLinkedList to construct Elements
            private Element(E data) {
                this.data = data;
                this.next = null;
            }

            public E getData() {
                return data;
            }

            public Element getNext() {
```

```
        public Element getNext() {
            return next;
        }

        private SinglyLinkedList getOwner() {
            return SinglyLinkedList.this;
        }
    }

    private Element head;
    private Element tail;
    private int size;

    public Element getHead() {
        return head;
    }

    public Element getTail() {
        return tail;
    }

    public int getSize() {
        return size;
    }

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

    public Element insertHead(E data) {
        Element newElement = new Element(data);

        if (isEmpty()) {
            // Insert into empty list
            head = newElement;
            tail = newElement;
        } else {
            // Insert into non-empty list
            newElement.next = head;
            head = newElement;
        }

        ++size;

        return newElement;
    }

    public Element insertTail(E data) {
        Element newElement = new Element(data);

        if (isEmpty()) {
            // Insert into empty list
            head = newElement;
            tail = newElement;
        } else {
```

```
            // Insert into non-empty list
            tail.next = newElement;
            tail = newElement;
        }

        ++size;

        return newElement;
    }

    public Element insertAfter(Element element, E data)
            throws IllegalArgumentException {
        // Check pre-conditions
        if (element == null) {
            throw new IllegalArgumentException(
                    "Argument 'element' must not be null");
        }
        if (element.getOwner() != this) {
            throw new IllegalArgumentException(
                    "Argument 'element' does not belong to this list");
        }

        // Insert new element
        Element newElement = new Element(data);
        if (tail == element) {
            // Insert new tail
            element.next = newElement;
            tail = newElement;
        } else {
            // Insert into middle of list
            newElement.next = element.next;
            element.next = newElement;
        }

        ++size;

        return newElement;
    }

    public E removeHead() throws NoSuchElementException {
        // Check pre-conditions
        if (isEmpty()) {
            throw new NoSuchElementException("Cannot remove from empty list");
        }

        // Remove the head
        Element oldHead = head;
        if (size == 1) {
            // Handle removal of the last element
            head = null;
            tail = null;
        } else {
            head = head.next;
        }
```

```
        --size;

        return oldHead.data;
    }

    // Note that there is no removeTail.  This cannot be implemented
    // efficiently because it would require O(n) to scan from head until
    // reaching the item _before_ tail.

    public E removeAfter(Element element)
            throws IllegalArgumentException, NoSuchElementException {
        // Check pre-conditions
        if (element == null) {
            throw new IllegalArgumentException(
                    "Argument 'element' must not be null");
        }
        if (element.getOwner() != this) {
            throw new IllegalArgumentException(
                    "Argument 'element' does not belong to this list");
        }
        if (element == tail) {
            throw new IllegalArgumentException(
                    "Argument 'element' must have a non-null next element");
        }

        // Remove element
        Element elementToRemove = element.next;
        if (elementToRemove == tail) {
            // Remove the tail
            element.next = null;
            tail = element;
        } else {
            // Remove from middle of list
            element.next = elementToRemove.next;
        }

        --size;

        return elementToRemove.data;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        SinglyLinkedList<?> that = (SinglyLinkedList<?>) o;

        if (this.size != that.size) return false;

        // Return whether all elements are the same
        SinglyLinkedList<?>.Element thisElem = this.getHead();
        SinglyLinkedList<?>.Element thatElem = that.getHead();
        while (thisElem != null && thatElem != null) {
```

```
            while (thisElem != null && thatElem != null) {
                if (!thisElem.getData().equals(thatElem.getData())) {
                    return false;
                }
                thisElem = thisElem.getNext();
                thatElem = thatElem.getNext();
            }

            return true;
        }
    }

    /**
     * This class implements a chained hash table backed by a singly-linked list.
     *
     * @param <K>
     * @param <V>
     */
    static class ChainedHashTable<K, V> {
        // Table of buckets
        private SinglyLinkedList<KeyValuePair<K, V>>[] table;

        private int size;

        private double maxLoadFactor = 1.5;

        private int resizeMultiplier = 2;

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

        //@SuppressWarnings("unchecked")
        public ChainedHashTable(int buckets) {
            // 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 ChainedHashTable(int buckets, double maxLoadFactor, int resizeMultiplier) {
            this(buckets);
            this.maxLoadFactor = maxLoadFactor;
            this.resizeMultiplier = resizeMultiplier;
        }

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

        public int getSize() {
            return size;
        }
```

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

    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();
        }

        // Has the load factor exceeded the threshold?
        if (((double) getSize() / table.length) > maxLoadFactor) resizeTable();

        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();
```

```
        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 illegalArgumentException) {
            return false;
        } catch (NoSuchElementException noSuchElementException) {
            return false;
        }

        return true;
    }

    private SinglyLinkedList<KeyValuePair<K, V>> getBucket(K key) {
        // Division method
        // return table[Math.abs(key.hashCode()) % table.length];
        // Multiplication method
        return table[(int) (table.length * (0.61 * (int) key % 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 void resizeTable() {
        SinglyLinkedList<KeyValuePair<K, V>>[] newTable = new SinglyLinkedList[table.length * resizeMultiplier];
        SinglyLinkedList<KeyValuePair<K, V>>[] originalTable = table;
        for (int index = 0; index < newTable.length; index++) {
            newTable[index] = new SinglyLinkedList<>();
        }
        this.table = newTable;
        SinglyLinkedList<KeyValuePair<K, V>>.Element element;
        for (int index = 0; index < originalTable.length; index++) {
            element = originalTable[index].head;
            while (element != null) {
                K key = element.getData().getKey();
                V value = element.getData().getValue();
                //table[Math.abs(key.hashCode()) % table.length].insertHead(new KeyValuePair<>(key, value));
                table[(int) (table.length * (0.61 * (int) key % 1))].insertHead(new KeyValuePair<>(key,
                        value));
                element = element.getNext();
            }
        }
    }

    @Override
    public String toString() {
        return String.format("Buckets %d; Elements %d; Load factor %.1f; Maximum load factor %.1f; Multiplier %d",
                table.length,
                getSize(),
                (double) getSize() / table.length,
                maxLoadFactor,
                resizeMultiplier);
    }
```

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

    /**
     * Extends RuntimeException instead of Exception since that's the convention set by NoSuchElementException.
     */
    static class DuplicateKeyException extends RuntimeException {
        public DuplicateKeyException() {
        }

        public DuplicateKeyException(String message) {
            super(message);
        }

        public DuplicateKeyException(Throwable cause) {
            super(cause);
        }

        public DuplicateKeyException(String message, Throwable cause) {
            super(message, cause);
        }
    }

    public static class Driver {
        public static void main(String[] args) {
            ChainedHashTable<Integer, String> chainedHashTable = new ChainedHashTable<>(3, 1.5, 4);
            System.out.println(chainedHashTable);
            System.out.println("Adding Kevin Cole, David Cole and Kamilah Cole…");
            chainedHashTable.insert(123456789, "Kevin Cole");
            chainedHashTable.insert(234567890, "David Cole");
            chainedHashTable.insert(345678901, "Kamilah Cole");
            System.out.println(chainedHashTable);
            System.out.println("Adding Cainan Cole, Adrienne Davis and Jahna Houston…");
            chainedHashTable.insert(567890123, "Cainan Cole");
            chainedHashTable.insert(678901234, "Adrienne Davis");
            chainedHashTable.insert(789012345, "Jahna Houston");
            System.out.println(chainedHashTable);
            System.out.println("Removing David Cole and Adrienne Davis…");
            chainedHashTable.remove(234567890);
            chainedHashTable.remove(678901234);
            System.out.println(chainedHashTable);
        }
    }

}
```

Sample Output (from Driver::main)

```
Buckets 3; Elements 0; Load factor 0.0; Maximum load factor 1.5; Multiplier 4
Adding Kevin Cole, David Cole and Kamilah Cole…
Buckets 3; Elements 3; Load factor 1.0; Maximum load factor 1.5; Multiplier 4
Adding Cainan Cole, Adrienne Davis and Jahna Houston…
Buckets 12; Elements 6; Load factor 0.5; Maximum load factor 1.5; Multiplier 4
Removing David Cole and Adrienne Davis…
Buckets 12; Elements 4; Load factor 0.3; Maximum load factor 1.5; Multiplier 4

Process finished with exit code 0
```

Homework5Test

```
package cse41321.containers;

import cse41321.containers.Homework5.ChainedHashTable;
import cse41321.containers.Homework5.KeyValuePair;
import cse41321.containers.Homework5.SinglyLinkedList;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

import static org.testng.Assert.*;

public class Homework5Test {

    @BeforeMethod
    public void setUp() {
    }

    @AfterMethod
    public void tearDown() {
    }

    @Test
    public void testTheKeyValuePairConstructor() {
        KeyValuePair<Integer, String> keyValuePair = new KeyValuePair<>(456789012, "Edgar Cole");
        // I'm not sure why I expected that in the next statement some sort of auto-unboxing would occur.
        assertEquals((int) keyValuePair.getKey(), 456789012);
        assertEquals(keyValuePair.getValue(), "Edgar Cole");
    }

    @Test
    public void testTheSinglyLinkedListConstructor() {
        KeyValuePair<Integer, String> keyValuePair = new KeyValuePair<>(456789012, "Edgar Cole");
        SinglyLinkedList<KeyValuePair<Integer, String>> singlyLinkedList = new SinglyLinkedList<>();
        singlyLinkedList.insertHead(keyValuePair);
        assertEquals(singlyLinkedList.getHead().getData().getValue(), "Edgar Cole");
        // Here we go again with the explicit cast. I guess I still have a bit to learn about how generics work.
        assertEquals((int) singlyLinkedList.getHead().getData().getKey(), 456789012);
    }
```

```
    }

    @Test
    public void theChainedHashTableConstructorTestBeforeModification() {
        ChainedHashTable<Integer, String> chainedHashTable = new ChainedHashTable<>(11);
        chainedHashTable.insert(456789012, "Edgar Cole");
        assertTrue(chainedHashTable.contains(456789012));
        chainedHashTable.insert(123456789, "Adrienne Davis");
        assertEquals(chainedHashTable.lookup(123456789), "Adrienne Davis");
        assertEquals(chainedHashTable.getSize(), 2);
        chainedHashTable.remove(456789012);
        assertEquals(chainedHashTable.getSize(), 1);
        assertEquals(chainedHashTable.lookup(123456789), "Adrienne Davis");
    }

    @Test
    public void theModifiedConstructorForTheChainedHashTable() {
        ChainedHashTable<Integer, String> chainedHashTable = new ChainedHashTable<>(11, 1.5, 2);
        assertTrue(chainedHashTable.isEmpty());
        chainedHashTable.insert(456789012, "Edgar Cole");
        assertFalse(chainedHashTable.isEmpty());
    }

    @Test
    public void theResizeMethod() {
        ChainedHashTable<Integer, String> chainedHashTable = new ChainedHashTable<>(7, 1.5, 4);
        assertTrue(chainedHashTable.isEmpty());
        chainedHashTable.insert(123456789, "Kevin Cole");
        chainedHashTable.insert(234567890, "David Cole");
        chainedHashTable.insert(345678901, "Kamilah Cole");
        chainedHashTable.insert(456789012, "Edgar Cole");
        assertEquals(chainedHashTable.getSize(), 4);
        assertEquals(chainedHashTable.lookup(345678901), "Kamilah Cole");
        assertEquals(chainedHashTable.getBuckets(), 7);
        chainedHashTable.resizeTable();
        assertEquals(chainedHashTable.getBuckets(), 28);
        assertEquals(chainedHashTable.getSize(), 4);
        assertEquals(chainedHashTable.lookup(345678901), "Kamilah Cole");
        assertEquals(chainedHashTable.lookup(123456789), "Kevin Cole");
    }

    @Test
    public void autoResizing() {
        ChainedHashTable<Integer, String> chainedHashTable = new ChainedHashTable<>(3, 1.5, 4);
        assertTrue(chainedHashTable.isEmpty());
        chainedHashTable.insert(123456789, "Kevin Cole");
        chainedHashTable.insert(234567890, "David Cole");
        chainedHashTable.insert(345678901, "Kamilah Cole");
        assertEquals(chainedHashTable.lookup(345678901), "Kamilah Cole");
        assertEquals(chainedHashTable.getBuckets(), 3);
        chainedHashTable.insert(567890123, "Cainan Cole");
        assertEquals(chainedHashTable.getSize(), 4);
        assertEquals(chainedHashTable.getBuckets(), 3);
        chainedHashTable.insert(678901234, "Adrienne Davis");
        assertEquals(chainedHashTable.getBuckets(), 3);
```

```
        chainedHashTable.insert(789012345, "Jahna Houston");
        assertEquals(chainedHashTable.getBuckets(), 12);
        assertEquals(chainedHashTable.lookup(678901234), "Adrienne Davis");
        assertEquals(chainedHashTable.getSize(), 6);
    }
}
```

Test Results

```
================================================
Default Suite
Total tests run: 6, Passes: 6, Failures: 0, Skips: 0
================================================


Process finished with exit code 0
```

Summary

Wow, this was a doozy! I had more trouble with the resize method than I had anticipated. In fact, I had expected it to be relatively easy. Instead, I was up until 6:00 in the morning getting it to work! Eventually I discovered that I had been creating a new array without initializing its elements with **SinglyLinkedlist** objects. That would explain the null pointer exceptions I kept getting. The sad part of it is that there was an example of what I needed to do elsewhere in the code! ☹

The next most challenging aspect of the assignment was deriving a formula for a hash function that would be based on multiplication rather than division. There were some type-casting issues I had to master before it worked.

I still think that generics are biting me in the butt. I need to get a better understanding of that. Sometimes I get lucky and guess right.

Each time I do one of these I'm making better use of the TestNG framework. I can't say the same about the IDEA debugger. I'm still learning how to use it. In terms of helping me find a bug, it succeeds only about forty percent of the time. I suspect I'll get better at using it.

About halfway through the all-nighter I was ready to throw in the towel. I mean, for good! However, my perseverance was rewarded, and eventually I solved the riddle. My hope is that I will retain the lessons learned so that each successive project will benefit from the preceding.

Okay, my computer is going to shut down in about fifteen minutes. Unfortunately, I don't know how to cancel the shutdown process on a Windows system.[1] I've got to scramble to get this submitted.

---

1. This situation has since been rectified.