Tanner Crane CPTS 233

I put all the code in here so I can save it as a PDF to upload to blackboard.

Github URL: https://github.com/Cybernetic1001/CPTS-233.git

```
/*
 *  Microassignment: Probing Hash Table addElement and removeElement
 *
 *  MA4_main: Main testing class for the hash table
 *   Tests inserts and deletes to evaluate implementation
 *
 *  Contributors:
 *    Bolong Zeng <bzeng@wsu.edu>, 2018
 *    Aaron S. Crandall <acrandal@wsu.edu>, 2019
 *
 *  Copyright:
 *   For academic use only under the Creative Commons
 *   Attribution-NonCommercial-NoDerivatives 4.0 International License
 *   http://creativecommons.org/licenses/by-nc-nd/4.0
 */

import java.util.Vector;

public class MA_HashTable_Add_Remove_main
{

	public static void main(String[] args) {
		small_demo();
		int testing_result = run_tests();
```

```
		if(testing_result > 0 ) {
			System.out.println("Some tests didn't pass.");
		}else{
			System.out.println("All tests passed - SUCCESS!.");
		}
		System.exit(testing_result);		// 0 means all tests pass
	}

	// Suite of tests on our Hash Table implementation
	public static int run_tests() {
		System.out.println(" ----- Beginning Hash Table Tests ----- ");
		int return_code = 0;

		return_code |= test_isEmpty();
		return_code |= test_size_whenEmpty();
		return_code |= test_addElement();
		return_code |= test_removeElement();
		return_code |= test_add_rem_add();
		return_code |= test_remove_nonexistent();

		return return_code;
	}

	public static int test_isEmpty() {
		int return_code = 0;
		System.out.print("Test: isEmpty() interface -- ");
		LinearHashTable<String, String> ht = new LinearHashTable<>(new
SimpleStringHasher());
		if( ht.isEmpty() == false ) {
```

```
                System.out.println(" FAIL");
                return_code = 1;            // Error, why isn't it empty?
            } else {
                System.out.println(" PASS");
            }
            return return_code;
        }

        public static int test_size_whenEmpty() {
            int return_code = 0;
            System.out.print("Test: size() when empty interface -- ");
            LinearHashTable<String, String> ht = new LinearHashTable<>(new
SimpleStringHasher());
            if( ht.size() != 0 ) {
                System.out.println(" FAIL");
                return_code = 1;            // Error, why isn't it empty?
            } else {
                System.out.println(" PASS");
            }
            return return_code;
        }

        public static int test_addElement() {
            int return_code = 0;
            System.out.println("Test: addElement() interface -- ");
            LinearHashTable<String, String> ht = new LinearHashTable<>(new
SimpleStringHasher());
            ht.addElement("KeyOne", "Element");
            ht.addElement("KeyTwo", "More Data");
```

```
        ht.addElement("AnotherKey", "Yet More Data");
        ht.printOut();

        System.out.print("  Should not be an empty table  -- ");
        if( ht.isEmpty() == true ) {
                System.out.println(" FAIL");
                return_code++; // Error, why isn't it empty?
        } else {
                System.out.println(" PASS");
        }
        System.out.print("  Should have 3 in table  -- ");
        if( ht.size() != 3 ) {
                System.out.println(" FAIL");
                return_code++; // Error, why isn't it empty?
        } else {
                System.out.println(" PASS");
        }

        System.out.println(" Doing deep data structure tests after add ");
        Vector<HashItem<String, String>> items = ht.getItems();
        return_code |= test_items_status(items, 4, "KeyOne", "Element", false);
        return_code |= test_items_status(items, 6, "KeyTwo", "More Data", false);
        return_code |= test_items_status(items, 7, "AnotherKey", "Yet More Data", false);

        return return_code;
}

public static int test_removeElement() {
        int return_code = 0;
```

```
		System.out.println("Test: removeElement() interface -- ");
		LinearHashTable<String, String> ht = new LinearHashTable<>(new
SimpleStringHasher());
		ht.addElement("KeyOne", "Element");
		ht.addElement("KeyTwo", "More Data 1");
		ht.addElement("KeyTwoM", "More Data 2");
		ht.addElement("KeyTwoMM", "More Data 3");
		ht.addElement("KeyTwoMMM", "More Data 4");
		ht.addElement("AnotherKey", "Yet More Data");
		ht.removeElement("KeyOne");			// Finds at hash key location
		ht.removeElement("KeyTwoM");	// Takes one hop to find
		ht.removeElement("AnotherKey");			// Needs a few hops to locate properly -
over KeyTwoM's delete
		ht.printOut();

		System.out.println(" Doing deep data structure tests after remove");
		Vector<HashItem<String, String>> items = ht.getItems();
		return_code |= test_items_status(items, 4, "KeyOne", "Element", true);
		return_code |= test_items_status(items, 6, "KeyTwo", "More Data 1", false);
		return_code |= test_items_status(items, 7, "KeyTwoM", "More Data 2", true);
		return_code |= test_items_status(items, 10, "AnotherKey", "Yet More Data", true);
		return_code |= test_items_status(items, 0, null, null, true);

		return return_code;
	}

	public static int test_add_rem_add() {
		int return_code = 0;
		System.out.println("Test: Add, Remove, Add same key with value update -- ");
```

```
        LinearHashTable<String, String> ht = new LinearHashTable<>(new
SimpleStringHasher());
        ht.addElement("KeyTwo", "More Data 1");
        ht.addElement("AnotherKey", "Yet More Data");
        ht.removeElement("AnotherKey");
        ht.addElement("AnotherKey", "Updated Data");
        Vector<HashItem<String, String>> items = ht.getItems();
        return_code |= test_items_status(items, 7, "AnotherKey", "Updated Data", false);
        ht.printOut();

        return return_code;
    }

    public static int test_remove_nonexistent() {
        int return_code = 0;
        System.out.println("Test: Add, remove non-existent key -- ");
        LinearHashTable<String, String> ht = new LinearHashTable<>(new
SimpleStringHasher());
        ht.addElement("KeyTwo", "More Data 1");
        ht.addElement("AnotherKey", "Yet More Data");
        ht.removeElement("NonExistentKey");
        Vector<HashItem<String, String>> items = ht.getItems();
        return_code |= test_items_status(items, 6, "KeyTwo", "More Data 1", false);
        return_code |= test_items_status(items, 7, "AnotherKey", "Yet More Data", false);
        ht.printOut();

        return return_code;
    }
```

```
	public static int test_items_status(Vector<HashItem<String, String>> items, int bucket, String key, String value, boolean status) {
		int return_code = 0;
		System.out.print("  Testing items[" + bucket + "] for key: " + key + " | val: " + value + " | deleted? " + status + " -- ");
		HashItem<String, String> slot = items.elementAt(bucket);
		if( slot.getKey() != key || slot.getValue() != value || slot.isEmpty() != status ) {
			return_code++;
			System.out.println("FAIL");
		} else {
			System.out.println("PASS");
		}

		return return_code;
	}


	// Small demo of hash table operating
  public static void small_demo()
  {
		System.out.println(" ----- Small Demo of Table Operating ----- ");
    LinearHashTable<String, String> ht = new LinearHashTable<>(new SimpleStringHasher());
    ht.addElement("I", "Love");
	ht.addElement("CptS", "233");
	ht.addElement("And", "I");
	ht.addElement("especially", "love");
		ht.addElement("Hashtables", "!");

		ht.printOut();                          // Dump the table out
```

```
    }
}
/*
 *  Microassignment: Probing Hash Table addElement and removeElement
 *
 *  SimpleStringHasher: Provides an object able to hash strings
 *
 *  Contributors:
 *    Bolong Zeng <bzeng@wsu.edu>, 2018
 *    Aaron S. Crandall <acrandal@wsu.edu>, 2019
 *
 *  Copyright:
 *   For academic use only under the Creative Commons
 *   Attribution-NonCommercial-NoDerivatives 4.0 International License
 *   http://creativecommons.org/licenses/by-nc-nd/4.0
 */


class SimpleStringHasher extends HasherBase<String>
{
    public int getHash(String key, int tableSize)
    {
        int hash = 0;
        for (char ch: key.toCharArray())
        {
            hash += ch;
        }
        return hash % tableSize;
    }
```

```
}
/*
 *  Microassignment: Probing Hash Table addElement and removeElement
 *
 *  LinearHashTable: Yet another Hash Table Implementation
 *
 *  Contributors:
 *    Bolong Zeng <bzeng@wsu.edu>, 2018
 *    Aaron S. Crandall <acrandal@wsu.edu>, 2019
 *
 *  Copyright:
 *   For academic use only under the Creative Commons
 *   Attribution-NonCommercial-NoDerivatives 4.0 International License
 *   http://creativecommons.org/licenses/by-nc-nd/4.0
 */


class LinearHashTable<K, V> extends HashTableBase<K, V>
{
        // Linear and Quadratic probing should rehash at a load factor of 0.5 or higher
   private static final double REHASH_LOAD_FACTOR = 0.5;

   // Constructors
   public LinearHashTable()
   {
      super();
   }

   public LinearHashTable(HasherBase<K> hasher)
```

```
{
   super(hasher);
}

public LinearHashTable(HasherBase<K> hasher, int number_of_elements)
{
   super(hasher, number_of_elements);
}

// Copy constructor
public LinearHashTable(LinearHashTable<K, V> other)
{
   super(other);
        }



// ***** MA Section Start ********************************************** //

// Concrete implementation for parent's addElement method
public void addElement(K key, V value)
{
   // Check for size restrictions
   resizeCheck();

   // Calculate hash based on key
   int hash = super.getHash(key);

   int original_hash = hash; // starting point for hashing
```

```
        while (true) {
            HashItem<K, V> h = super.getItems().elementAt(hash);

            if (h.isEmpty()) {
                super.getItems().elementAt(hash).setKey(key);
                super.getItems().elementAt(hash).setValue(value);  //Updating the attributes according to parameters of HashItem if empty spot is found
                super.getItems().elementAt(hash).setIsEmpty(false);
                super._number_of_elements++;
                break;
            }

            hash++; //Increment through the hashtable

            if (hash >= super.getItems().capacity())
            {
              hash = 0;
            }
            else if(hash == original_hash)
            {
                System.out.println("Empty cell not found"); // State empty if a new element is not found
                return;

            }
        }


    }
        // MA TODO: find empty slot to insert (update HashItem as necessary)
```

```
   // Remember how many things we are presently storing (size N)
        //  Hint: do we always increase the size whenever this function is called?
   //_number_of_elements++;



// Removes supplied key from hash table
public void removeElement(K key)
{
   // Calculate hash from key
   int hash = super.getHash(key);
   int original_hash = hash;
   boolean found = false;
   HashItem<K, V> h = super.getItems().elementAt(hash); //declare h

while(!found)
{


  if(h.getKey() == key) // check if key is @ hash
  {

   found = true;
   super.getItems().elementAt(hash).setIsEmpty(true); // remove element from table
   super._number_of_elements--; // reduce the number of elements



  }
  hash++;
```

```
  if (hash  >= super.getItems().capacity()) //Ensure hash doesn't exceed bounds
  {
     hash = 0;
  }

  h = super.getItems().elementAt(hash); // if not, jump to next key


   if(original_hash == hash)  // if our hash is equal to starting hash then key does not exist
   {
      break;
   }

}

}


   // MA TODO: find slot to remove. Remember to check for infinite loop!
   //  ALSO: Use lazy deletion - see structure of HashItem


   // Make sure decrease hashtable size
        //  Hint: do we always reduce the size whenever this function is called?
   // _number_of_elements--;

// ***** MA Section End ********************************************** //
```

```
// Public API to get current number of elements in Hash Table
        public int size() {
                return this._number_of_elements;
        }

// Public API to test whether the Hash Table is empty (N == 0)
        public boolean isEmpty() {
                return this._number_of_elements == 0;
        }

// Returns true if the key is contained in the hash table
public boolean containsElement(K key)
{
   int hash = super.getHash(key);
   HashItem<K, V> slot = _items.elementAt(hash);

   // Left incomplete to avoid hints in the MA :)
   return false;
}

// Returns the item pointed to by key
public V getElement(K key)
{
   int hash = super.getHash(key);
   HashItem<K, V> slot = _items.elementAt(hash);

   // Left incomplete to avoid hints in the MA :)
```

```
        return null;
    }

    // Determines whether or not we need to resize
    //  to turn off resize, just always return false
    protected boolean needsResize()
    {
        // Linear probing seems to get worse after a load factor of about 50%
        if (_number_of_elements > (REHASH_LOAD_FACTOR * _primes[_local_prime_index]))
        {
            return true;
        }
        return false;
    }

    // Called to do a resize as needed
    protected void resizeCheck()
    {
        // Right now, resize when load factor > 0.5; it might be worth it to experiment with
        //  this value for different kinds of hashtables
        if (needsResize())
        {
            _local_prime_index++;

            HasherBase<K> hasher = _hasher;
            LinearHashTable<K, V> new_hash = new LinearHashTable<K, V>(hasher,
_primes[_local_prime_index]);

            for (HashItem<K, V> item: _items)
```

```
        {
            if (item.isEmpty() == false)
            {
                // Add element to new hash table
                new_hash.addElement(item.getKey(), item.getValue());
            }
        }

        // Steal temp hash object's internal vector for ourselves
        _items = new_hash._items;
    }
  }

  // Debugging tool to print out the entire contents of the hash table
        public void printOut() {
                System.out.println(" Dumping hash with " + _number_of_elements + " items in " +
_items.size() + " buckets");
                System.out.println("[X] Key       | Value | Deleted");
                for( int i = 0; i < _items.size(); i++ ) {
                        HashItem<K, V> curr_slot = _items.get(i);
                        System.out.print("[" + i + "] ");
                        System.out.println(curr_slot.getKey() + " | " + curr_slot.getValue() + " | " +
curr_slot.isEmpty());
                }
        }
}
/*
 *  Microassignment: Probing Hash Table addElement and removeElement
 *
```

```
 *  IntegerHasher: Provides an object able to hash Integers
 *
 *  Contributors:
 *    Bolong Zeng <bzeng@wsu.edu>, 2018
 *    Aaron S. Crandall <acrandal@wsu.edu>, 2019
 *
 *  Copyright:
 *   For academic use only under the Creative Commons
 *   Attribution-NonCommercial-NoDerivatives 4.0 International License
 *   http://creativecommons.org/licenses/by-nc-nd/4.0
 */



class IntegerHasher extends HasherBase<Integer>
{
   public int getHash(Integer key, int tableSize)
   {
      return key % tableSize;
   }
}


/*
 *  Microassignment: Probing Hash Table addElement and removeElement
 *
 *  HashTableBase: Base class for different kinds of Hash Tables
 *   Defines common features and API
 *
 *  Contributors:
 *    Bolong Zeng <bzeng@wsu.edu>, 2018
```

```
 *    Aaron S. Crandall <acrandal@wsu.edu>, 2019
 *
 *  Copyright:
 *   For academic use only under the Creative Commons
 *   Attribution-NonCommercial-NoDerivatives 4.0 International License
 *   http://creativecommons.org/licenses/by-nc-nd/4.0
 */

/*
Considerations when implementing a Hash Table:
        * How do we convert the key into an integer value (the hash)?
        * What happens when we try to store a KVP in an already full slot (collision)?
        * What are appropriate sizes of the underlying vector?
        * How does performance change as our underlying vector fills up?
                * When is it necessary to resize?
                        * How do we resize?

*/

import java.util.Vector;
abstract class HashTableBase<K, V>
{
        // Note: Java uses 11 as the starting hashtable size.
        protected Vector<HashItem<K, V>> _items;
        protected static int _primes[] = { 11, 47, 97, 197, 379, 691, 1259, 2557, 5051, 7919, 14149,
28607, 52817, 102149, 209939, 461017, 855683, 1299827 };
        protected static int _primes_count = _primes.length;
        protected HasherBase<K> _hasher;
```

```
// Keeps track of our position in our prime index counter
//  Used when we rehash
protected int _local_prime_index;
protected int _number_of_elements;

// Checks to see whether or not it's time to resize and rehashes
protected abstract void resizeCheck();

// Shortcut method for calling full hasher's getHash function
//  This is a form of a wrapper or facade software design pattern
//  See "Gang of Four" book on design patterns
protected int getHash(K item)
{
        return _hasher.getHash(item, _items.size());
}

public HashTableBase()
{

}

public HashTableBase(HasherBase<K> hasher)
{
        _hasher = hasher;
        _items = initItems(11);
        _local_prime_index = 0;
        _number_of_elements = 0;

        while (_primes[_local_prime_index] < 11)
```

```
		{
			_local_prime_index ++;
		}
	}

	public HashTableBase(HasherBase<K> hasher, int number_of_elements)
	{
		_hasher = hasher;
		_items = initItems(number_of_elements); //new Vector<HashItem<K,
V>>(number_of_elements);
		_local_prime_index = 0;
		_number_of_elements = 0;

		while (_primes[_local_prime_index] < number_of_elements)
		{
			_local_prime_index ++;
		}
	}

	// Init an _items vector
	protected Vector<HashItem<K, V>> initItems(int number_of_elements)
	{
		Vector<HashItem<K,V>> v_hash = new Vector<HashItem<K,V>>(number_of_elements);
		for (int i = 0; i < v_hash.capacity(); i ++)
		{
			v_hash.add(new HashItem<K, V>());
		}
		return v_hash;
	}
```

```
// Copy constructor
public HashTableBase(HashTableBase<K, V> other)
{
	_hasher = other._hasher;
	_items = other._items;			// This is *BADNESS* can you see why?
	_local_prime_index = other._local_prime_index;
	_number_of_elements = other._number_of_elements;
}

// Determines whether or not a given key exists in the hash table
public boolean hasKey(K key)
{
	try
	{
		V result = getElement(key);
		return true;
	}
	catch (Exception e)
	{
		//e.printStackTrace();	// Uncomment for deep debugging issues
		return false;
	}
}

// Returns all keys present in the hashtable
//  Class Question: How can we make this more efficient?
public Vector<K> getKeys()
{
```

```
        Vector<K> keys = new Vector<>();

        // Iteration through _items vector
        for (HashItem<K, V> item: _items)
        {
                if (item.isEmpty() == false)
                {
                        keys.add(item.getKey());
                }
        }
        return keys;
}

public Vector<HashItem<K, V>> getItems()
{
        return _items;
}

// TODO: Implement remove function based on desired collision mechanism
public abstract void removeElement(K key);

// TODO: Implement contains check based on desired collision mechanism
public abstract boolean containsElement(K key);

// TODO: Implement getElement method based on desired collision mechanism
public abstract V getElement(K key);

// TODO: Implement add function based on desired collision mechanism
public abstract void addElement(K key, V value);
```

```
}

/*
 *  Microassignment: Probing Hash Table addElement and removeElement
 *
 *  HashItem: Hash Table storage wrapper class
 *   Allows us to do lazy deletion
 *
 *  Contributors:
 *    Bolong Zeng <bzeng@wsu.edu>, 2018
 *    Aaron S. Crandall <acrandal@wsu.edu>, 2019
 *
 *  Copyright:
 *   For academic use only under the Creative Commons
 *   Attribution-NonCommercial-NoDerivatives 4.0 International License
 *   http://creativecommons.org/licenses/by-nc-nd/4.0
 */

class HashItem<K, V> {
        private boolean _is_empty = true;
        private V _item = null;
        private K _key = null;

        public HashItem()
        {
                //HashItem new_one = new HashItem();
        }

        public HashItem(K key, V value, boolean is_empty)
```

```
{
        this._key = key;
        this._item = value;
        this._is_empty = is_empty;
}

public HashItem(K key, V value)
{
        this._key = key;
        this._item = value;
}

public V getValue()
{
        return _item;
}

public void setValue(V item)
{
        this._item = item;
}

public K getKey()
{
        return _key;
}

public void setKey(K key)
{
```

```
            this._key = key;
        }

        public boolean isEmpty()
        {
            return _is_empty;
        }

        //Note: use this function wisely
        public boolean isTrueEmpty()
        {
            return _is_empty && _key == null && _item == null;
        }

        public void setIsEmpty(boolean value)
        {
            this._is_empty = value;
        }
}

/*
 *  Microassignment: Probing Hash Table addElement and removeElement
 *
 *  HasherBase: Base class for Hash Functions
 *   Provides a class that can be overriden to do different hash algorithms
 *   Structured to do a form of passed in function handling
 *   Designed to make the implementation of hash algorithms to be more dynamic
 *
 *  Contributors:
```

```
 *    Bolong Zeng <bzeng@wsu.edu>, 2018
 *    Aaron S. Crandall <acrandal@wsu.edu>, 2019
 *
 *  Copyright:
 *    For academic use only under the Creative Commons
 *    Attribution-NonCommercial-NoDerivatives 4.0 International License
 *    http://creativecommons.org/licenses/by-nc-nd/4.0
 */



// A base class to provide a general interface for all hash functions
// To use this, you build your own Hashing Class that inherits from HasherBase
// Then, you implement your own getHash and then pass your object to the HashTable class
// This is a form of treating the Hasher as a first class function:
//  https://en.wikipedia.org/wiki/First-class_function
// I would consider this the "Strategy" design pattern from the Gang of Four book:
//  https://springframework.guru/gang-of-four-design-patterns/strategy-pattern/
abstract class HasherBase<T>
{
        public abstract int getHash(T item, int mod_by);
}


// Notes:
//
//HasherBase<String> hb = new HasherBase<>();
//All the subclasses of the abstract class, has to implement its defined abstract functions/operations
//
//Define many other Hashers based upon this base:
// IntegerHasher
```

```
// StringHasher

// DoubleHasher

// MyOwnClassHasher
```