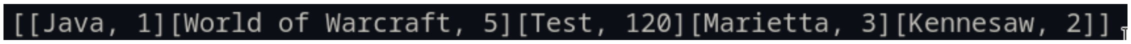
Assignment 9: Hashing

CS3305/W01 Data Structures

Casey Hampson

October 29, 2024

Program Output



```
[[Java, 1][World of Warcraft, 5][Test, 120][Marietta, 3][Kennesaw, 2]]
```

Source Code

```java
// Name: Casey Hampson
// Class: CS 3305/W01
// Term: Fall 2024
// Instructor: Sharon Perry
// Assignment: 9-Hashing

import java.util.HashSet;
import java.util.Set;


class MyHashMap<K,V> implements MyMap<K,V> {
    // define some defaults
    private static final int INITIAL_CAPACITY = 4;
    private static final double LOAD_FACTOR_THRESHOLD = 0.5;
    private static final int MAXIMUM_CAPACITY = 1 << 16;

    // the current capacity and size (num of elements)
    private int capacity = INITIAL_CAPACITY, size = 0;

    // arraylist of entries in the hashmap
    Entry<K,V>[] table;

    /**
     * The constructor will simply initialize the array of entries
     * with the initial capacity we have defined above
     */
    MyHashMap() {
        table = new Entry[INITIAL_CAPACITY];
    }


    /**
     * Sets all the elements in the array to null
     */
    @Override
    public void clear() {
        this.size = 0;
        for (int i=0; i<this.capacity; i++) {
            this.table[i] = null;
        }
    }


    /**
     * Returns true if the given key is present in the HashMap; false otherwise
     */
    @Override
    public boolean containsKey(K key) {
        return (get(key) != null);
    }
```

```java
/**
 * Returns true if the given value is present in the HashMap; false otherwise
 */
@Override
public boolean containsValue(V value) {
    for (Entry<K,V> entry : this.table) {
        if (entry == null) continue;
        if (entry.getValue().equals(value)) return true;
    }

    return false;
}


/**
 * Returns a set of all the (non-null) entries in the HashMap
 */
@Override
public Set<Entry<K,V>> entrySet() {
    Set<Entry<K,V>> entry_set = new HashSet<>();

    for (Entry<K,V> entry : this.table) {
        if (entry == null) continue;
        entry_set.add(entry);
    }

    return entry_set;
}



/**
 * Returns the value associated with the given key,
 * or null if the key is not present in the HashMap
 */
@Override
public V get(K key) {
    // grab the index for the requested key in the table
    int table_idx = hash(key.hashCode());
    if (this.table[table_idx] == null) return null;

    // we must ensure that there is no collision with
    // key hashes matching despite keys not being equal
    // so we check the actual key itself rather than just
    // pulling the key at the corresponding hash/index
    Entry<K,V> entry = this.table[table_idx];
    if (entry.getKey().equals(key)) return entry.getValue();
    return null;
}



/**
 * Returns whether or not the HashMap is empty
 */
@Override
public boolean isEmpty() {
    return (this.size == 0);
}



/**
 * Returns a set of all the keys in the HashMap
 */
@Override
public Set<K> keySet() {
    Set<K> key_set = new HashSet<>();
    for (Entry<K,V> entry : this.table) {
        if (entry == null) continue;
        key_set.add(entry.getKey());
    }
    return key_set;
}


/**
 * Adds the (key, value) pair to the hash map
 *
 * This is where the biggest part of the assignment lies
 * if we have a collision, in order to implement linear probing,
 * we simply need to continue moving forward in the array
 * until we find an empty slot
 * There will always be one as long as we ensure that
 * the load factor is < 0.5
 */
@Override
public V put(K key, V value) {
    // first check that the key is not already in the table
    // if it is, we replace the old value with this new one
    // and return the old one
    if (get(key) != null) {
        int table_idx = hash(key.hashCode());
        Entry<K,V> entry = this.table[table_idx];
        V old_val = entry.getValue();
        entry.value = value;
        return old_val;
    }

    // otherwise, we need to check the load factor
    // and if we are at (or above) the load factor, then we rehash
    if (this.size >= this.capacity * LOAD_FACTOR_THRESHOLD) rehash();

    // at this point, sizes and such are fine
    // and we are ready to insert
    // if we get a collision,
    // we push forward in the array until there is no collision
    int table_idx = hash(key.hashCode());
```

```
        while (this.table[table_idx] != null) table_idx = (table_idx+1) % this.capacity;
        // the `table_idx` should now be pointing at the next available spot
        table[table_idx] = new Entry<>(key, value);

        // then just increment the size
        // and return the value to indicate a success
        this.size++;
        return value;
    }



    /**
     * Removes the key (and associated value) from the HashMap
     */
    @Override
    public void remove(K key) {
        int table_idx = hash(key.hashCode());
        this.table[table_idx] = null;
        this.size--;
    }


    /**
     * Returns the number of entries current in the HashMap
     */
    @Override
    public int size() {
        return this.size;
    }



    /**
     * Returns a set of all the values in the HashMap
     */
    @Override
    public Set<V> values() {
        Set<V> vals = new HashSet<>();
        for (Entry<K,V> entry : this.table) {
            if (entry == null) continue;
            vals.add(entry.getValue());
        }
        return vals;
    }



    @Override
    public String toString() {
        StringBuilder builder = new StringBuilder("[");

        for (Entry<K,V> entry : this.table) {
            if (entry == null) continue;
            builder.append(entry);
        }

        builder.append("]");
        return builder.toString();
    }


    /**
     * A simplified generating of the hash
     * takes the hash code and simply takes its
     * modulus with the capacity of the table
     */
    private int hash(int hashCode) {
        // i found some hashcodes give back negative numbers, so jsut in case
        return Math.abs(hashCode) & (this.capacity-1);
    }

    /**
     * Resizes the table and rehashes all of the entries
     */
    private void rehash() {
        // first grab all the entries
        // and ``reset'' the table
        Set<Entry<K,V>> entry_set = entrySet();
        this.capacity <<= 1;
        this.table = new Entry[this.capacity];
        this.size = 0;

        // then re add everything to this new table
        for (Entry<K,V> entry : entry_set) put(entry.getKey(), entry.getValue());
    }


}


public class A9 {
    public static void main(String[] args) {
        MyHashMap<String, Integer> map = new MyHashMap<>();
        map.put("Java", 1);
        map.put("Kennesaw", 2);
        map.put("Marietta", 3);
        map.put("World of Warcraft", 5);
        map.put("Test", 120);
        System.out.println(map);
    }
}
```