Resolución Segundo Parcial de Programación Orientada a Objetos (72.33)

10/06/2020

Ejercicio 1

```java
@FunctionalInterface
public interface Mapper<ValueIn, ValueOut> {

   ValueOut map(ValueIn valueIn);

}
```

```java
public interface ReversibleMap<K,V> extends Map<K,V> {

   ReversibleMap<V,K> reverse();

   <T> ReversibleMap<T,K> reverse(Mapper<V,T> mapper);


}
```

```java
public class ReversibleMapImpl<K,V> extends HashMap<K,V> implements
ReversibleMap<K,V> {

   @Override
   public ReversibleMap<V, K> reverse() {
       return reverse(v -> v);
   }

   @Override
   public <T> ReversibleMap<T, K> reverse(Mapper<V, T> mapper) {
       ReversibleMap<T,K> map = new ReversibleMapImpl<>();
       for(Map.Entry<K,V> element : entrySet()) {
           map.put(mapper.map(element.getValue()), element.getKey());
       }
       return map;
   }

}
```

Ejercicio 2

Opción 1

```java
public class LimitedCache<K,V> implements Cache<K,V> {

   private int maxSize;
   private Map<K,V> cache = new HashMap<>();
   private Map<K, Integer> hits = new HashMap<>();

   public LimitedCache(int maxSize) {
       if(maxSize <= 0) {
            throw new IllegalArgumentException();
        }
        this.maxSize = maxSize;
    }

    @Override
    public void add(K key, V value) {
        if(!cache.containsKey(key) && size() == maxSize) {
            removeLessAccessEntry();
        }
        cache.put(key, value);
        hits.putIfAbsent(key, 0);
    }

    private void removeLessAccessEntry() {
        K least = hits.entrySet().stream()
                .min(Comparator.comparing(Map.Entry::getValue))
                .get()
                .getKey();
        cache.remove(least);
        hits.remove(least);
    }

    @Override
    public V get(K key) {
        V value = cache.get(key);
        if(hits.containsKey(key)) {
            hits.merge(key, 1, Integer::sum);
        }
        return value;
    }

    @Override
    public int size() {
        return cache.size();
    }

}
```

Opción 2

```
public class LimitedCache<K,V> implements Cache<K,V> {

    private int maxSize;
    private Map<K,AccessValue> cache = new HashMap<>();

    public LimitedCache(int maxSize) {
        if(maxSize <= 0) {
            throw new IllegalArgumentException();
        }
        this.maxSize = maxSize;
    }

    @Override
    public void add(K key, V value) {
        if(cache.containsKey(key)) {
            cache.get(key).value = value;
            return;
        }
        if(size() == maxSize) {
            removeLessAccessEntry();
        }
        cache.put(key, new AccessValue(value));
    }

    private void removeLessAccessEntry() {
        Map.Entry<K, AccessValue> aux = cache.entrySet().iterator().next();
        for(Map.Entry<K, AccessValue> entry : cache.entrySet()) {
            if(entry.getValue().access < aux.getValue().access) {
                aux = entry;
            }
        }
        K least = aux.getKey();
        cache.remove(least);
    }

    @Override
    public V get(K key) {
        if(!cache.containsKey(key)) {
            return null;
        }
        AccessValue accessValue = cache.get(key);
        accessValue.addAccess();
        return accessValue.value;
    }

    @Override
    public int size() {
        return cache.size();
    }

    private class AccessValue {

        private V value;
        private int access;

        public AccessValue(V value) {
            this.value = value;
        }

        void addAccess() {
            access++;
        }

    }

}
```

Ejercicio 3

```
class QuotaFinalExam < FinalExam

 def initialize(name, quota)
   super(name)
   @quota = quota
 end

 def enroll(student)
   return super(student) unless enrolled_count >= @quota
   puts "Error Enrolling #{student}"
 end

end
```

```
class CorrelativeFinalExam < FinalExam

 def initialize(name, correlatives)
   super(name)
   @correlatives = correlatives
 end

 def enroll(student)
   return super(student) if correlatives_for_exam?(student)
   # tambien podia ser if (@correlatives - student.approved_courses).empty?
   puts "Error Enrolling #{student}"
 end

 private def correlatives_for_exam?(student)
   @correlatives.each do |correlative|
     return false unless student.approved?(correlative)
   end
   true
 end

end
```

```
class Student

 attr_reader :name

 def initialize(name, approved_courses)
   @name, @approved_courses = name, approved_courses
 end

 def ==(other)
   return false unless other.is_a?(Student)
   @name == other.name
 end

 def to_s
   "#{@name}"
 end

 def inspect
   to_s
 end

 def approved?(course_name)
   @approved_courses.include?(course_name)
 end
```

```
end
```

```
class Course

 attr_reader :name

 def initialize(name)
   @name = name
 end

 def ==(other)
   return false unless other.is_a?(Course)
   @name == other.name
 end

end
```