Data Structures: Sixth Recitation Questions (With solutions)

1. What is the worst-case running time for inserting n key-value pairs into an initially empty map M that is implemented with the `UnsortedMap` class?

   Solution. The worst-case running time for inserting n key-value pairs into an initially empty map M implemented with the `UnsortedMap` class is $O(n^2)$. This is because for each insertion, the map must potentially scan through all existing entries to ensure no duplicate key is inserted, leading to a quadratic time complexity as the number of entries grows.

2. The use of null values in a map is problematic, as there is then no way to differentiate whether a null value returned by the call `get(`k`)` represents the legitimate value of an entry (k, null), or designates that key k was not found. The `java.util.Map` interface includes a boolean method, `containsKey(`k`)`, that resolves any such ambiguity. Implement such a method for the `UnsortedMap` class.

   Solution. To implement the `containsKey(k)` method in the `UnsortedMap` class, iterate through all the entries in the map. If an entry with the key k is found, return `true`. If no such entry is found after checking all entries, return `false`. This method allows distinguishing between a `null` value associated with an existing key and the absence of a key in the map.

3. Draw the 11-entry hash table that results from using the hash function, $h(i) = (3i + 5) \mod 11$, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by chaining.

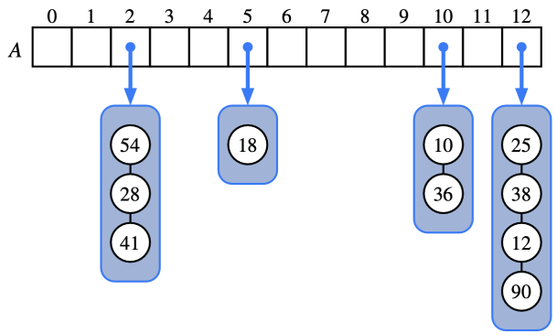
   Solution. Omit. Did this in recitation.

4. What is the result of the previous exercise, assuming collisions are handled by linear probing?

   Solution. Omit. Did this in recitation.

5. What is the worst-case time for putting n entries in an initially empty hash table, with collisions resolved by chaining? What is the best case?

   Solution. The worst case is similar to Problem 1. The best case, assuming a perfect hash function and uniform distribution, is $O(1)$ for each insertion, as each entry goes to a different slot with no collisions. So the total time is $O(N)$



6. Show the result of rehashing the hash table shown in the figure above into a table of size 19 using the new hash function $h(k) = 3k \mod 17$.

   Solution. Omit

7. Consider the goal of adding entry (k, v) to a map only if there does not yet exist some other entry with key k. For a map M (without null values), this might be accomplished as follows.

   ```
   if (M.get(k) == null)
       M.put(k, v);
   ```

   While this accomplishes the goal, its efficiency is less than ideal, as time will be spent on the failed search during the get call, and again during the put call (which always begins by trying to locate an existing entry with the given key). To avoid this inefficiency, some map implementations support a custom method `putIfAbsent(`k`, `v`)` that accomplishes this goal. Give such an implementation of `putIfAbsent` for the `UnsortedTableMap` class. Do not look at the `put` method.

   Solution. To implement `putIfAbsent(k, v)` in the `UnsortedTableMap` class, first check if the key k already exists in the map by iterating through all entries. If k is not found, insert the new key-value pair. This method avoids the inefficiency of a redundant search and insert operation by ensuring that each key is unique upon insertion.

8. Describe how to perform a removal from a hash table that uses linear probing to resolve collisions where we do not use a special marker to represent deleted elements. That is, we must rearrange the contents so that it appears that the removed entry was never inserted in the first place.

   Solution. To perform a removal from a hash table that uses linear probing without special markers for deleted elements, remove the target element and then rehash all subsequent elements in the probe sequence. This ensures that all elements are still findable and maintains the integrity of the probing sequence.

9. Write a function that takes a list of strings as input, where each string can be considered a word in the document. The function should analyze this document and return a map that maps each unique word found in the document to its number of occurrences.

   Solution. To write a function that analyzes a document and returns a map of each unique word to its number of occurrences, iterate through the list of strings, breaking each string into words. For each word, increment its count in the map if it already exists, or add it with a count of 1 if it does not. This results in a map where keys are unique words and values are the counts of their occurrences.

10. An **inverted file** is a critical data structure for implementing applications such as an index of a book or a search engine. Given a document D, which can be viewed as an unordered, numbered list of words, an inverted file is an ordered list of words, L, such that, for each word w in L, we store the indices of the places in D where w appears. Design an efficient algorithm for constructing L from D.

    Solution. To design an algorithm for constructing an inverted file from a document D, iterate through each word in the document, recording its index each time it appears. Store this information in a data structure where each word is associated with a list of its indices in D. Sort this list by the words to create the ordered list L, ensuring efficient lookup and retrieval of word locations within the document.

```
public class UnsortedMap<K, V> {
    private int size;
    private GoodList<Entry<K, V>> entrylist;

    private class UnsortEntry<K, V> implements Entry<K, V> {
        K key;
        V value;

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

        public K getKey();
        public V getValue();
    }

    public UnsortedMap();

    int size();
    boolean isEmpty();
    GoodList<Entry<K,V>> entrySet();
```

```
    GoodList<K> keySet();
    GoodList<V> values();

    V put(K key, V value);
    V get(K key);
    V remove(K key);
}
```