Hashing

TJHSST

March 2, 2004

1 Background

TreeMaps organize data into balanced Binary Search Trees where each node stores both a key and its associated value, and the nodes are ordered based on the keys. Iteration over the keySet provides access to the data as an in-order walk of the keys. Since the BST is always balanced, the `put` and `get` operations are both order O(log n).

2 Concept

Another approach is to store the data into an array. In the simplest case, data is stored in a lookup table by converting each key to an integer and storing the associated value in that particular cell of the array. Of course, this requires that each key produces a unique integer and that our array be as large as any integer that could be produced by any possible key.

HashMaps organize data into a hash table where each key is converted to an integer using the `hashCode` method. This integer is then further massaged to ensure uniform distribution within the size of the array. If two distinct keys produce the same integer, a collision, then chaining is used to place the multiple values into a bucket (usually a linked list).

The load factor, α, of the hash table is the average size of the buckets, which affects hashing's order O(1) promise. A set tolerance determines when the underlying hash table must be resized. For instance, the HashMap class provided by Sun has a default tolerance of 0.75 and an initial size of 16.

An alternative response to collisions is called probing. In this case, slots are continually chosen until an empty slot is found (thus, no buckets). In either system the `equals` method is ultimately used to determine when a key has been found. Note carefully that this is different from a TreeMap, in which the `compareTo` method is used exclusively.

3 Problems

Write Java programs to do each of the following:

1. Create all the length three permutations of the capital letters and the ten digits. There are 36^3 such strings. Determine each string's hash code and display, for each unique hash code, a list of all the strings that produce that particular hash code. Find the maximum size of such a list and all the hash codes that achieve that size.

2. Recursion can be extremely inefficient. Frequently this inefficiency results from performing a process or calculation many times on the same input. These redundant calculations could be avoided if we had a data structure to remember which instances of a calculation had been performed previously and the results of that previous calculation. Consider the following two methods:

```
public static long f(long n)
{
    return (n % 2 == 0) ? n / 2 : 3 * n + 1;
}
public static long g(long n)
{
    long count = 0, temp = n;
    do
    {
        ++count;
        temp = f(temp);
    }while(temp != 1);
    return count;
}
```

Notice that the value returned by `g` is the number of times that `f` must be iterated in order to produce the value 1. As far as is known, this will always occur. You are to use a HashMap to remember the values that have been previously calculated in order to avoid redundant calculations. Specifically, your program should accept a positive integer `n` as input and produce as output the values of `g(1)`, `g(2)`, through `g(n)` without any repeated evaluations.[1]

3. Compare the performance of a recursive version of the Fibonacci sequence to an implementation that uses an array to remember the previously calculated values. Use the `System.currentTimeMillis();` command to time your memoized version versus the standard program.

[1] Converted to Java from the original by Bill Lyle of Murray State.