Overview

- Hash Functions
- Java's Builtin Hash Tables

Hash Functions

Most hash functions assume that the universe of keys is the set of natural numbers. Thus, if the keys are not natural numbers, a way must be found to interpret them as such. For example, a key that is a character string can be interpreted as a natural number expressed in a radix notation. Thus, the identifier `pt` might be interpreted as the pair of natural numbers (112, 116), because `p`$= 112$ and `t`$= 116$ in the ASCII character set, which has 128 characters. Expressed as a radix-128 number, `pt` becomes $(112 \times 128) + 116 = 14452$.

It is usually straightforward in any given application to devise some such simple method for interpreting each key as a (possibly large) natural number.

The Division Method

This method maps a key k into one of m slots by taking the remainder of k divided by m, i.e.,

$$h(k) = k \ mod \ m \tag{1}$$

We usually avoid certain values of m, such as powers of 2. Good values for m are primes not too close to powers of 2.

The Multiplication Method

This method has two steps

1. Multiply the key k by a constant A in the range $0 < A < 1$ and extract the fractional part of kA.

2. Multiply this value by m and take the floor of the result.

In short, the hash function is

$$h(k) = \lfloor m(kA \ mod \ 1) \rfloor \tag{2}$$

where "$kA \ mod \ 1$" means the fractional part of kA, i.e., $kA - \lfloor kA \rfloor$.

The Multiplication Method (cont.)

An advantage of this method is that the value of m is not critical. It is typically chosen to be a power of 2—$m = 2^p$ for some integer p—since we can easily implement this function:

Suppose that the word size of the machine is w bits and that k fits into a word. We first multiply k by the w-bit integer $\lfloor 2^w A \rfloor$. The result is a $2w$-bit value $r_1 2^w + r_0$, where r_1 is the high-order word of the product and r_0 is the low-order word of the product. The desired p-bit hash value consists of the p most significant bits of r_0.

Hash Functions: Uses Beyond Hash Tables?

Suppose we wish to search a linked list of length n, where each element contains a key k along with a hash value $h(k)$. Each key is a *long* character string. How might we take advantage of the hash values when searching the list for an element with a given key?

Java's Builtin Hash Tables

Java's `java.util` package contains a hash table class. This `Hashtable` class extends the `Dictionary` (abstract) class.

A `Dictionary` is an object that associates elements with keys. Every key and every element is an object. In any one `Dictionary`, every key is associated at most one element. Given a `Dictionary` and a key, the associated element can be looked up.

Java's `Dictionary` Class

The `Dictionary` class is defined as follows.

```
public abstract class Dictionary {
    abstract public int size();
    abstract public boolean isEmpty();
    abstract public Object get(Object key)
            throws NullPointerException;
    abstract public Object put(Object key,
                               Object element)
            throws NullPointerException;
    abstract public Object remove(Object key)
            throws NullPointerException;
    abstract public Enumeration keys();
    abstract public Enumeration elements();
}
```

If the key (or element passed to `put`) is null, a NullPointerException is thrown. If the key passed to `get` is not found, null is returned.

Java's Hashtable Class

The Hashtable class is defined as follows.

```
public class Hashtable extends Dictionary
                       implements Cloneable {
    public Hashtable(int initialCapacity,
                     float loadFactor);
    public Hashtable(int initialCapacity);
    public Hashtable(); /* load 0.75 */
    public String toString();
    public Object clone();
    public int size();
    public boolean isEmpty();
    public Object get(Object key)
            throws NullPointerException;
    public Object put(Object key, Object value)
            throws NullPointerException;
    public Object remove(Object key)
            throws NullPointerException;
    public Enumeration keys();
```

Java's `Hashtable` Class (cont.)

```
    public Enumeration elements();
    public boolean contains(Object value);
    public boolean containsKey(Object key);
    protected void rehash();
    public void clear();
}
```

A `Hashtable` has two parameters that affect its efficiency: its capacity and its load factor. The load factor should be between 0.0 and 1.0. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is increased, using the rehash method. Larger load factors use memory more efficiently at the expense of larger expected time per lookup. If many entries are to be made in a hash table, creating it with a sufficiently large capacity may allow the entries to be inserted more efficiently than letting it perform automatic rehashing as needed to grow the table.

Java's `Object` Class's `hashCode` Method

Java's `Object` Class defines a `hashCode` Method.

```
public class Object {
    public final Class getClass();
    public String toString();
    public boolean equals(Object obj);
    public int hashCode();
    ...
```

This method is supported principally for the benefit of hash tables such as those provided by the Java library class java.util.Hashtable.