Overview

- Hash Tables
- Hash Functions

Hash Tables

A hash table is a generalization of an ordinary array.

When the number of keys actually stored is small relative to the total number of possible keys, hash tables become an effective alternative to directly addressing an array, since a hash table typically uses an array of size proportional to the number of keys actually stored.

Instead of using the key as an array index directly, the array index is *computed* from the key.

Hash Tables (cont.)

With hashing, an element with key k is stored in slot $h(k)$; i.e., a **hash function** h is used to compute the slot from the key k.

h maps the universe U of keys into the slots of a **hash table** $T[0..m-1]$:

$$h : U \rightarrow \{0, 1, ..., m-1\} \tag{1}$$

The Problem: Collisions

Two keys may hash to the same slot. This is called a **collision**. Because $|U| > m$, collisions are unavoidable.

To avoid collisions, h should appear “random”, i.e., adjacent keys should not hash to adjacent slots.

To cope with collisions, the simplest method is **chaining**.

Chaining

In chaining, we put all the elements that hash to the same slot in a linked list, i.e., slot j contains a reference to the head of the list of all stored elements that hash to j; if there are no such elements, slot j contains `null`.

To insert an element, we simply put it at the front of the list. So, the worst case running time is $O(1)$.

To lookup an element, we search the list belonging to the slot for the corresponding key. So, the worst case running time is proportional to the length of the list.

Removal is identical to lookup.

Performance

Given a hash table with m slots that stores n elements, we define the **load factor** α as n/m, i.e., the average number of elements in a chain.

The worst case behavior of hashing with chaining is $O(n)$: All n keys hash to the same slot, creating a list of length n.

The expected case behavior depends on how well the hash function distributes the set of keys to be stored among the m slots, on average. We will assume that (1) any given element is equally likely to hash into any of the m slots and (2) the hash value can be computed in $O(1)$ time. Then the expected case search time is $O(1 + \alpha)$.

Performance (cont.)

If the number of hash table slots is at least proportional to the number of elements in the table, we have $n = O(m)$ and consequently, $\alpha = n/m = O(m)/m = O(1)$. Thus, searching takes constant time on average.

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.,

$$h(k) = k \; mod \; m \tag{2}$$

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

$$h(k) = \lfloor m(kA \ mod \ 1) \rfloor \tag{3}$$

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.