Hashing

Today, we will talk about hash tables, hash functions, and methods of dealing with collisions.

Hash Tables

A hash table is a data structure supporting the abstract data type of a (dynamic) set, with the following operations:

- INSERT(k, v): insertion of a key k with corresponding value v.
- SEARCH(k): search for a key k.
- REMOVE(k): remove a key k from the hash table.

Hash Functions

A hash function is a function h that maps keys k to slots in an array; that is, for each key k, $h(k)$ is in the range $0, 1, \ldots, m-1$, where m is the size of the hash table.

We want our keys to be distributed approximately randomly over $0, 1, \ldots, m-1$, because that minimizes the chances of keys' being hashed to the same value. A more precise version of this statement is called the Simple Uniform Hashing Assumption, or SUHA. Also, we want our hash function to take $O(1)$ time to calculate for each key and be deterministic, so that we may look up a key after hashing it.

Examples of hash functions:

- $h(k) = k \mod m$
- $h(k) = \lfloor m(kA \mod 1) \rfloor$ for $A \approx (\sqrt{5}-1)/2$
- $h(s) = [\text{sum of ascii characters}] \mod m$
- $h(s) = [\text{sum of ascii 4-byte blocks}] \mod m$
- $h(k) = (A \cdot k) \mod 2^w >> (w-r)$, that is, bits r through w of $A \cdot k$.
- $h(k) = ((ak+b) \mod p) \mod m$.

What are advantages and disadvantages of these hash functions? (Possible answers: time to compute, programmer time, chance of collisions.)

Implementing a Hash Table

We need to implement our basic operations:

- INSERT(k, v): hash k to an index in the table; place k along with v at that location (or produce an error if the table is full). Runtime $O(1)$.
- SEARCH(k): hash k to an index; check whether k is there or not. If it is there, return the corresponding value v. Runtime $O(1)$.
- REMOVE(k): hash k to an index; remove (k, v) pair at that index if present. Runtime $O(1)$.

Collisions

We did not consider collisions in our analysis. If we assume the SUHA, then collisions will be relatively rare; probability $1/m$ that two given keys hash to the same slot. Despite this, they will still happen, and we have to handle them somehow.

The easiest method for handling collisions is **chaining**. With chaining, each slot in our hash table is actually a linked list. Our operations become:

- INSERT(k, v): hash k to an index in the table; add k along with v to the linked list at that location. Runtime $O(1)$.
- SEARCH(k): hash k to an index; scan linked list for key k. Runtime $O(l)$, where l is length of linked list.
- REMOVE(k): hash k to an index; scan linked list for key k and remove if present. Runtime $O(l)$.

But wait! We wanted runtime $O(1)$, not $O(l)$. What is l? If our hash table has n elements and m slots, then $l = n/m$ on average. We define $\alpha = n/m$ to be the "load" for our hash table, and $l = O(\alpha)$ on average. If we keep our load low, we can have $l = O(1)$ on average, keeping constant (expected) time operations. Because we cannot choose n, we must choose m so that $m = \Theta(n)$ on average. This involves resizing our table.

Later, we will learn about other techniques for collision resolution, including open addressing.

Problems and Applications

We now solve a few problems involving hashing.

Zero-Sum Game

Suppose that we have a list of n numbers. How can we detect if two of them add to a target value t?

Hint: Consider hashing the numbers.

Now suppose that the numbers are randomly generated 32-bit integers. Can you solve this problem using sorting? Which approach might be best in practice, and why?

Mod 9

Demonstrate what happens when we insert the keys $5, 28, 19, 15, 20, 33, 12, 17, 10$ into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \mod 9$.