csce750 — Analysis of Algorithms
Fall 2019 — Lecture Notes: Hash Tables

This document contains slides from the lecture, formatted to be suitable for printing or individual reading, and with some supplemental explanations added. It is intended as a supplement to, rather than a replacement for, the lectures themselves — you should not expect the notes to be self-contained or complete on their own.

1 Introduction

CLRS 11

A **hash table** is a specific kind of data structure implementing these operations:

- INSERT(k)
- SEARCH(k)
- DELETE(k)

The **keys** are drawn from a **universe** $\mathcal{U}$:

$$\mathcal{U} = \{0, 1, \ldots, |\mathcal{U}| - 1\}$$

(We are skipping CLRS 10, on "Elementary Data Structures.")

2 A simple solution: Direct address tables

We could use an array T of size $|\mathcal{U}|$, and store the element with key k at $T[k]$.

Advantage: Fast. All operations $\Theta(1)$ time. Very simple.

Disadvantage: Requires lots of memory if $\mathcal{U}$ is large.

3 Hashing

Key idea: Instead of storing k at $T[k]$, choose an efficiently computable **hash function**:

$$h : \mathcal{U} \to \{0, \ldots, m - 1\},$$

with $m < |\mathcal{U}|$.

Store k at $T[h(k)]$.

(Intuition: The hash function must be deterministic, but should be "random-looking.")

4 Building a hash function

Creating hash functions is not a precise science, but there are some useful patterns.

- **Division method**:
$$h(k) = k \mod m$$
- **Multiplication method**:
$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

With good choices for m and A, both of these tend to produce good approximations of the **simple uniform hashing assumption**:

> For each k we insert and each $j = 0, \ldots, m-1$,
> $$Pr(h(k) = j) = \frac{1}{m}.$$

This leads to an **average case** analysis, because it starts with an assumption about the inputs we'll get.

5 Example: Python hash function for string keys

```
static long string_hash(PyStringObject *a)
{
    register Py_ssize_t len;
    register unsigned char *p;
    register long x;

    if (a->ob_shash != -1)
        return a->ob_shash;
    len = Py_SIZE(a);
    p = (unsigned char *) a->ob_sval;
    x = *p << 7;
    while (--len >= 0)
        x = (1000003*x) ^ *p++;
    x ^= Py_SIZE(a);
    if (x == -1)
        x = -2;
    a->ob_shash = x;
    return x;
}
```

6 Collisions

Because $m < |\mathcal{U}|$, we must be prepared for **collisions**, which occur when we insert two distinct keys k_1 and k_2, with $h(k_1) = h(k_2)$.

There are two primary choices for resolving collisions:

- **Chaining** — Use a secondary data structure (linked list, hash table, etc.) for each element of T.
- **Probing** — If $h(k)$ is occupied, try somewhere else.

7 Analysis: Hashing with linked-list chaining

Searching, worst case: Collision every time. $\Theta(n)$ time.

> *Note: Insert is faster:* $\Theta(1)$.

8 Analysis: Hashing with linked-list chaining

Searching, under simple uniform hashing assumption: For an unsuccessful search:

$$\begin{aligned} A(n) &= \Theta(1) + \sum_{i=0}^{m-1} \frac{1}{m} \operatorname{length}(T[i]) \\ &= \Theta(1) + \frac{1}{m} \sum_{i=0}^{m-1} \operatorname{length}(T[i]) \\ &= \Theta(1) + \frac{n}{m} \\ &= \Theta\left(1 + \frac{n}{m}\right) \end{aligned}$$

For a successful search: Similar, but messier. (CLRS 259)

$$\Theta\left(1 + \frac{n}{m}\right)$$

If $n = O(m)$ — table size proportional to number of keys — then this is $O(1 + cm/m) = O(1)$ time.

9 A word about probing

In the probing method, the table itself stores everything, without any other data structures.

Simple example: If $h(k)$ is occupied, try $h(k)+1$, then $h(k)+2$, then $\ldots$. (For better choices, see CLRS 272.)

Advantage: Can be more memory efficient, because *all* of the available memory is devoted to the table. No pointers.

Disadvantage: Deleting is harder. (Why?)