Hashing (Chapter 5)

The hash table ADT supports insertions, deletions, and finds in **constant average time**.

Basic idea: map each item to a number and use that number as an index into a hash table data structure.

We will:

- analyze methods for implementing hash tables
- compare hash tables with binary search trees

Skip Section 5.6.

Hashing Operations

- FIND(item)

- INSERT(item): Check if the item is in the table and if not, place it there.

- DELETE(item): If item is in the table, delete it.

What operations are slow ($O(n)$ time)?

What makes the operation non-trivial is **collisions:** Two items hash to the same location in the table.

What's another possible problem?

Hash Functions

Data structure is an array of size H_SIZE.

Each item has a **key** which we assume to be a string.

The key is mapped to a number by a *hash function*:

- The hash function should be quick to compute.
- Distinct keys should get different cells as much as possible. (See Figure 5.1)

What is a simple hash function if the key is an integer?

What happens if the table size is 10 and all keys end in zero?

What is a simple hash function in the case that the key is a string?

What happens if the table size is 10,007 and the keys are ≤ 8 characters long?

What is wrong with the following hash function?

```
function hash(key: string_type;
              key_size: integer): INDEX;
begin
  hash := (ord(key[1]) + 27 * ord(key[2])
           + 729 * ord(key[3])) (mod H_SIZE);
end;
```

See Figure 5.5 for a hash function that does not suffer from these problems.

Remaining problem: how to handle collisions. Always check whether a cell is full before inserting an item.

What do you do if it's full?

Open Hashing

Idea: keep a linked list at each location in the hash table which stores each element hashed to that location. (See Figure 5.6)

What is the worst case behavior of this approach?

Should the linked lists be sorted?

$$\text{Load factor} = \frac{\text{number of elements in the hash table}}{\text{table size}}$$

Heuristic: the table size should be made so that the load factor is approximately 1.

Closed Hashing

A different way of resolving collisions called "closed" because the hashed items remain in the table!

If a collision occurs, alternate cells are tried until an empty cell is found.

More formally, cells $h_0(x), h_1(x), h_2(x), \ldots$ are tried in succession where

$$h_i(x) = (hash(x) + f(i)) \pmod{H\_SIZE}$$

with $f(0) = 0$.

The big question: which alternate cells do you try?

Linear Probing

Keep trying the next adjacent cell.
If a cell is full, always go "next door."

$$f(i) = i$$

- How do you find items in a hash table using linear probing?
- What is the problem with linear probing? (See Fig. 5.11)
- What happens under linear probing if the load factor is close to 1?
- Why not just use a random number generator to determine where to place a key?

Quadratic Probing

$$f(i) = i^2$$

The goal is to solve the *primary clustering* problem of linear probing. (See Fig. 5.13)

The problem of *secondary clustering* remains: each succesive item hashed to a location has to explore the same sequence of offset locations.

Good news: A new element can always be inserted if the table is at least half-empty and the table size is prime.

Bad news: There is no guarantee of finding an empty cell once the table is more than half-full!

If we use quadratic probing on a table of size 10, will we end up trying each of the slots in the table?

Proof of Good News Theorem

Let H_SIZE be an (odd) prime greater than 3.

Lemma: The first $\lfloor H\_SIZE/2 \rfloor$ alternate locations are all distinct.

Proof: (by contradiction)
Consider $h(x) + i^2 \pmod{H\_SIZE}$
and $h(x) + j^2 \pmod{H\_SIZE}$,
where $0 < i, j \leq \lfloor H\_SIZE/2 \rfloor$.

Suppose, for the sake of contradiction, that these locations are the same, but that $i \neq j$. Then

$$\begin{aligned}
h(x) + i^2 &= h(x) + j^2 \pmod{H_SIZE} \\
i^2 &= j^2 \pmod{H_SIZE} \\
i^2 - j^2 &= 0 \pmod{H_SIZE} \\
(i-j)(i+j) &= 0 \pmod{H_SIZE}
\end{aligned}$$

Since H_SIZE is prime, it follows that either $(i-j)$ or $(i+j)$ is equal to $0 \pmod{H\_SIZE}$. Why does this lead to contradiction?

If $i - j = 0$, then $i = j$
and if $i + j \pmod{H\_SIZE} = 0$, then
$i + j = k * H\_SIZE$,
but both i and j are $\leq H\_SIZE/2$.

We have established the lemma, now we go to prove the Good News Theorem.

Proof: if the table is at least half-empty, then there are at least $H\_SIZE/2 + 1$ locations free.

Since the quadratic function will generate at least $H\_SIZE/2$ alternatives, it must find some free location in the table.

(Consider painting the free locations blue, and painting red the locations that the hash function will try. Is it possible that no location will be painted both red and blue? By the pigeon-hole principle the answer is: no)

Double Hashing

The problem with both linear and quadratic probing is that once a new element hashes to the same value as some old elements, you follow the same sequence of alternate locations until finding a free cell. **Double hashing solves this problem.**

$$f(i) = i * h_2(x)$$

We apply a second hash function to x and probe at a distance of $h_2(x), 2h_2(x), \ldots,$

What happens if $h_2(x)$ evaluates to zero?

What's wrong with $h_2(x) = x \bmod 9$?

A better choice is $h_2(x) = 7 - (x \bmod 7)$.

Important: The hash table size ought to be prime!

Example and Comparison

The Example Table in Fig. 5.18 has size 10. Let's see what happens if we try to insert 23 into that table.

$$h_2(23) = 7 - 2 = 5$$

23 would collide with 58 in location 3, and the success of hash values are 5, 0, 5, 0, ….

	BINARY TREES	**HASH TABLES**
FIND		
INSERT/Delete		
FIND-MINIMAL-ELEMENT		

What operations are supported by a binary search tree, but not by a hash table?

Applications of Hash Tables

Compilers: Keeping track of declared variables in source code (a symbol table).

Operations are INSERTs and FINDs, and identifiers are typically short.

Game-playing programs: Keeping track of previously-seen positions to avoid recomputing the best move (a transposition table).

On-line spelling checker? By hashing a dictionary, we can tell whether a particular word is misspelled. However, a good spelling checker also suggests a possible correction to the misspelled word. Could you do that with a hash table?

Conclusion: Hash tables are extremely fast data structures for a limited set of operations.