Problem

Large database of key-value pairs.

Given a key, you need to find the corresponding value as fast as possible.
Example: Phone book lookup
Example: fb account
Example: Bank balances

Balance Inquiry
Chk Acct_0206
Available Balance $8,112.96
Present Balance $8,112.96
Insert(A,x)

Search(A,x)

Delete(A,x)
Chapter 11 Hash Tables

11.1 Direct-address tables

Direct addressing is a simple technique that works well when the universe $U$ of keys is reasonably small. Suppose that an application needs a dynamic set in which each element has a key drawn from the universe $U = \{0, 1, \ldots, m\}$, where $m$ is not too large. We shall assume that no two elements have the same key.

To represent the dynamic set, we use an array, or direct-address table, denoted by $T[0..m]$ in which each position, or slot, corresponds to an element in the universe $U$. Figure 11.1 illustrates this approach; slot $k$ points to an element in the set with key $k$. If the set contains no element with key $k$, then $T[k]$ contains NIL.

The dictionary operations are trivial to implement:

- **DIRECT-ADDRESS-SEARCH**.
  \[ T[k] \]
  \[ T[k] \text{ contains } \begin{cases} \text{key} & \text{if } k \in K \text{ and } k \text{ has not been deleted}, \\ \text{NIL} & \text{otherwise.} \end{cases} \]

- **DIRECT-ADDRESS-INSERT**.
  \[ T[k] := k \]

- **DIRECT-ADDRESS-DELETE**.
  \[ T[k] := \text{NIL} \]

Each of these operations takes only $O(1)$ time.

---

**First Idea**

Direct addressing: store key $X$ at index $X$.
Insert(A,x) \quad A[x] = x

Search(A,x) \quad \text{if } A[x] == x, \text{ return } x

Delete(A,x) \quad A[x] = \bot

Each operation is constant-time.
But space requirement is \(O(U)\).
PROBLEM 1 Arrays

Many algorithms use arrays as a data structure; and good programming practice requires us to first initialize an array before using it. The time to initialize or “zero-out” the array is usually proportional to the size of the array. In some cases, however, the time it takes to zero-out the array will overwhelm the actual running time of the algorithm. This problem develops a way to overcome this problem.

Suppose your algorithm needs to access an array \( A[1, \ldots, n] \) only \( k \) times during its execution. Design a method that “simulates” access to a zeroed array \( A \) using only \( O(k) \) time (thus, you cannot spend \( O(n) \) time to initialize \( A \) since \( k \) might be, say \( \sqrt{n} \) and therefore much smaller). By “simulate access,” we mean devise two functions \( \text{read}(i) \) and \( \text{write}(i, x) \) so that when you call \( \text{read} \) on an index \( i \) that has not already been set, it will read \( 0 \). When you call \( \text{read} \) on an index \( i \) that has been written to before, it returns the last written value. The array \( A \) may initially contain garbage (assume it is adversarially chosen garbage). You can also use extra memory in order to run your simulation (but this extra memory may also initially contain garbage). Both \( \text{read} \) and \( \text{write} \) should run in constant time.

problem 2  ICPC problems

Goal: K keys, O(K) space, but O(1) expected-time operations.

Idea: store key $k$ at location $h(k)$

$h$ is the hash function.
Problem: What happens if 2 keys map to the same position?

"collision"
First solution: Chaining

Keep a linked list of elements that hash to each position.
Insert(A, x): Add x to front of list at A[ h(x) ]

Search(A, x): Look for x in list at A[ h(x) ]

What is the running time of Insert?

O(1)

What is the running time of Search?
Worst-case:

All elements “hash” to the same bucket. $O(n)$ time to search
Universal hashing

Family of hash functions $H=\{h_i\}$

When you select one randomly from $H$, for any two keys $k_i, k_j$,

$$\Pr[ h_i(k_i) = h_i(k_j) ] \leq 1/m$$
Universal Hashing

\[ n_{h(k)} : \text{length of the list that contains key } k \]

\[ \mathbb{E} \left[ n_{h(k)} \right] \leq 1 + \frac{n}{m} \]

(lets do this on the board)
Universal hashing

$$\{ h_{a,b} \}$$

$$h_{a,b}(x) = (ax + b \mod p) \mod m$$
static int hash(int h) {
    // This function ensures that hashCode values that differ only by
    // constant multiples at each bit position have a bounded
    // number of collisions (approximately 8 at default load factor).
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}

public int hashCode() {
    int h = hash;
    if (h == 0) {
        int off = offset;
        char val[] = value;
        int len = count;

        for (int i = 0; i < len; i++) {
            h = 31*h + val[off++];
        }
    }

    hash = h;
    return h;
}
Big problem: balls & bins

Throw n balls into n bins.

Expected to get 1 ball per bin.

However, the probability that some bin gets “many balls” is quite noticeable.
Example: Birthdays

365 days. What is the probability that 2 people have the same birthday?
Suppose birthdays are chosen totally randomly!

\[
\begin{array}{cccccccc}
364 & 363 & 362 & 361 & 360 & \cdots & 350 \\
365 & 365 & 365 & 365 & 365 & \cdots & 365
\end{array}
\]
Pr n people have different birthdays

Pr

# people
Balls & Bins

Throw n balls into n bins.
Carter-Wegman hashing

First level hash

1. hash elements into buckets
2. hash elements in each bucket using another hash function.

Second level hashing uses array of size $n_j^2$
Pr[perfect hashing of $m$ elements into $m^2$ buckets]
$E \left[ \text{Sum } n_j^2 \right] < 2n$
\[ \Pr[ \text{total space} > 4n ] < 1/2 \]