CH 9.2 : HASH TABLES

ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH DATA STRUCTURES AND ALGORITHMS IN C++, GOODRICH, TAMASSIA AND MOUNT (WILEY 2004) AND SLIDES FROM NANCY M. AMATO AND JORY DENNY
Sometimes a key can be interpreted or transformed into an address. In this case, we can use an implementation called a **hash table** for the Map ADT.

**Hash tables**

- Essentially an array $A$ of size $N$ (either to an element itself or to a “bucket”)
- A **Hash function** $h(k) \rightarrow [0, N - 1]$, $h(k)$ is referred to as the **hash value**
  - Example - $h(k) = k \mod N$
- Goal is to store elements $(k, v)$ at index $i = h(k)$
ISSUES WITH HASH TABLES

- **Issues**
  - **Collisions** - some keys will map to the same index of H (otherwise we have a Direct Address Table).
    - Chaining - put values that hash to same location in a linked list (or a “bucket”)
    - Open addressing - if a collision occurs, have a method to select another location in the table.
  - **Load factor**
  - **Rehashing**
We design a hash table for a Map storing items (SSN, Name), where SSN (social security number) is a nine-digit positive integer.

Our hash table uses an array of size $N = 10,000$ and the hash function $h(k) = \text{last four digits of } k$. 

The hash function maps the SSNs to indices in the array, with the hash values shown for some entries.
A hash function is usually specified as the composition of two functions:

- **Hash code:**
  \[ h_1 : \text{keys} \rightarrow \text{integers} \]

- **Compression function:**
  \[ h_2 : \text{integers} \rightarrow [0, N - 1] \]

The hash code is applied first, and the compression function is applied next on the result, i.e.,
\[ h(k) = h_2(h_1(k)) \]

The goal of the hash function is to “disperse” the keys in an apparently random way.
HASH CODES

- **Memory address:**
  - We reinterpret the memory address of the key object as an integer
  - Good in general, except for numeric and string keys

- **Integer cast:**
  - We reinterpret the bits of the key as an integer
  - Suitable for keys of length less than or equal to the number of bits of the integer type (e.g., byte, short, int and float in C++)

- **Component sum:**
  - We partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows)
  - Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type (e.g., long and double in C++)
Polynomial accumulation:

- We partition the bits of the key into a sequence of components of fixed length (e.g., 8, 16 or 32 bits)
  \[ a_0 a_1 \ldots a_{n-1} \]

- We evaluate the polynomial
  \[ p(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_{n-1} z^{n-1} \]
  at a fixed value \( z \), ignoring overflows

- Especially suitable for strings (e.g., the choice \( z = 33 \) gives at most 6 collisions on a set of 50,000 English words)

Cyclic Shift:

- Like polynomial accumulation except use bit shifts instead of multiplications and bitwise or instead of addition

- Can be used on floating point numbers as well by converting the number to an array of characters
COMPRESSION FUNCTIONS

- **Division:**
  - $h_2(k) = |k| \mod N$
  - The size $N$ of the hash table is usually chosen to be a prime (based on number theory principles and modular arithmetic)

- **Multiply, Add and Divide (MAD):**
  - $h_2(k) = |ak + b| \mod N$
  - $a$ and $b$ are nonnegative integers such that $a \mod N \neq 0$
  - Otherwise, every integer would map to the same value $b$
Collisions occur when different elements are mapped to the same cell.

Separate Chaining: let each cell in the table point to a linked list of entries that map there.

Chaining is simple, but requires additional memory outside the table.
Assume you have a hash table $H$ with $N = 9$ slots ($A[0 – 8]$) and let the hash function be $h(k) = k \mod N$

Demonstrate (by picture) the insertion of the following keys into a hash table with collisions resolved by chaining

- 5, 28, 19, 15, 20, 33, 12, 17, 10
In **Open addressing** the colliding item is placed in a different cell of the table.

**Linear probing** handles collisions by placing the colliding item in the next (circularly) available table cell. So the \( i \)th cell checked is:

\[
h(k, i) = |h(k) + i| \mod N
\]

Each table cell inspected is referred to as a “probe”.

Colliding items lump together, causing future collisions to cause a longer **probe sequence**.

---

**Example:**

- \( h(k) = k \mod 13 \)
- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order.

![Probe sequence example](image-url)
SEARCH WITH LINEAR PROBING

- Consider a hash table $A$ that uses linear probing
- $\text{find}(k)$
  - We start at cell $h(k)$
  - We probe consecutive locations until one of the following occurs
    - An item with key $k$ is found, or
    - An empty cell is found, or
    - $N$ cells have been unsuccessfully probed

Algorithm $\text{find}(k)$

1. $i \leftarrow h(k)$
2. $p \leftarrow 0$
3. repeat
4. $c \leftarrow A[i]$
5. if $c \neq \emptyset$
6. return null
7. else if $c.\text{key}() = k$
8. return $c$
9. else
10. $i \leftarrow (i + 1) \mod N$
11. $p \leftarrow p + 1$
12. until $p = N$
13. return null
To handle insertions and deletions, we introduce a special object, called AVAILABLE, which replaces deleted elements.

- **erase**($k$)
  - We search for an item with key $k$
  - If such an item ($k, v$) is found, we replace it with the special item AVAILABLE

- **put**($k, v$)
  - We start at cell $h(k)$
  - We probe consecutive cells until one of the following occurs
    - A cell $i$ is found that is either empty or stores AVAILABLE, or
    - $N$ cells have been unsuccessfully probed
Assume you have a hash table $H$ with $N = 11$ slots ($A[0 \rightarrow 10]$) and let the hash function be $h(k) = k \mod N$.

Demonstrate (by picture) the insertion of the following keys into a hash table with collisions resolved by linear probing:
- 10, 22, 31, 4, 15, 28, 17, 88, 59
- Linear probing has an issue with clustering.
- Another strategy called quadratic probing uses a hash function 
  \[ h(k, i) = (h(k) + i^2) \mod N \]
  for \( i = 0, 1, ..., N - 1 \)
- This can still cause secondary clustering.
Double hashing uses a secondary hash function \( h_2(k) \) and handles collisions by placing an item in the first available cell of the series

\[
h(k, i) = (h_1(k) + ih_2(k)) \mod N
\]
for \( i = 0, 1, ..., N - 1 \)

- The secondary hash function \( h_2(k) \) cannot have zero values
- The table size \( N \) must be a prime to allow probing of all the cells

- Common choice of compression map for the secondary hash function:
  \[
h_2(k) = q - (k \mod q)
\]
  where
  - \( q < N \)
  - \( q \) is a prime
- The possible values for \( h_2(k) \) are \( 1, 2, ..., q \)
In the worst case, searches, insertions and removals on a hash table take $O(n)$ time.

The worst case occurs when all the keys inserted into the map collide.

The load factor $\lambda = \frac{n}{N}$ affects the performance of a hash table.

Assuming that the hash values are like random numbers, it can be shown that the expected number of probes for an insertion with open addressing is $\frac{1}{1 - \lambda} = \frac{1}{1 - \frac{n}{N}} = \frac{1}{N - \frac{n}{N}} = \frac{N}{N - n}$.

The expected running time of all the Map ADT operations in a hash table is $O(1)$.

In practice, hashing is very fast provided the load factor is not close to 100%.

Applications of hash tables:
- Small databases
- Compilers
- Browser caches
The **probe sequence** of a key $k$ is the sequence of slots probed when looking for $k$

- In open addressing, the probe sequence is $h(k, 0), h(k, 1), ..., h(k, N - 1)$

**Uniform Hashing Assumption**

- Each key is equally likely to have any one of the $N!$ permutations of $\{0, 1, ..., N - 1\}$ as is probe sequence
- Note: Linear probing and double hashing are far from achieving Uniform Hashing
  - Linear probing: $N$ distinct probe sequences
  - Double Hashing: $N^2$ distinct probe sequences
Theorem: Assuming uniform hashing and an open-address hash table with load factor \( \lambda = \frac{n}{N} < 1 \), the expected number of probes in an unsuccessful search is at most \( \frac{1}{1-\lambda} \).

Exercise: compute the expected number of probes in an unsuccessful search in an open address hash table with \( \lambda = \frac{1}{2}, \lambda = \frac{3}{4} \), and \( \lambda = \frac{99}{100} \).
ON REHASHING

- Keeping the load factor low is vital for performance
- When resizing the table:
  - Reallocate space for the array
  - Design a new hash function (new parameters) for the new array size
  - For each item you reinsert it into the table
## SUMMARY MAPS/DICTIONARIES (SO FAR)

<table>
<thead>
<tr>
<th>Method</th>
<th>put$(k,v)$</th>
<th>find$(k)$</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log File (map only)</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Direct Address Table</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(N)$</td>
</tr>
<tr>
<td>(map only)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lookup Table (ordered map/dict)</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Hashing (chaining)</td>
<td>$O(1)$</td>
<td>$O(n/N)$</td>
<td>$O(n + N)$</td>
</tr>
<tr>
<td>Hashing (open addressing)</td>
<td>$O\left(\frac{1}{1 - \frac{n}{N}}\right)$</td>
<td>$O\left(\frac{1}{1 - \frac{n}{N}}\right)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>