The hash-table data structure achieves (near) constant time searching by “wasting” memory space.

☞ the size of the memory we reserve for a hash table is typically much larger than the number of data stored in it.

• Each item in the table comprises a key and a value.

☞ In each GMU student record, we could use GMU ID as the key and all other data combined as the value.

• We use a hashing function to determine where to store items in table.

☞ Given an entry \( x \), it will be stored at location \( h(x, \text{key}) \), where \( h \) is the hashing function.

• The verb “to hash” means to chop something up or to make a mess out of it (it/something being the key).
Example

- The table contains 80 entries.
- The key of each entry is an integer.
- We use the hash function $h(key) = key \% 80$.
  - An entry with key 140 will be stored at location $140 \% 80 = 60$ in the table.
  - An entry with key 425 will be stored at location $425 \% 80 = 25$.
  - An entry with key 825 will be also stored at location $825 \% 80 = 25$.

Key-to-Index Mapping

- A **collision** occurs when two keys are “hashed” to the same location in the table.
- **Birthday paradox:** If 23 people are present in a room, chances are good ($> 0.5$) that two of them will have the same month/day birthday.
  - If we randomly maps 23 keys into a table of size 365, the probability that no two keys map to the same location is less than 0.5.
- A complete key-to-index mapping consists of
  1. a hashing function that minimizes collisions, and
  2. a technique for handling collisions.
Hashing Function I: Division

- Let CAPACITY be the number of entries in the table.
- \( h(key) = key \% \text{CAPACITY} \).
  - Please notice that this hash function is called “division,” although what we actually want is its remainder.
- Certain CAPACITY values are better than others at avoiding collisions.
- Studies show that a good choice of CAPACITY is a prime number of the form \( 4c + 3 \), where \( c \) is some integer number.
  - 811 is a prime number equal to \((4 \times 202) + 3\)
  - 1019 is a prime number equal to \((4 \times 254) + 3\)

Hashing Function II: Mid-Square

- The key is multiplied by itself.
- The hashing function returns middle \( r \) digits of the result.
- CAPACITY is set to \( 2^r \).
Hashing Function III: Multiplication

- The key is multiplied by a constant less than one.
- The hash function returns the first $r$ digits of the result.
- Again, CAPACITY is set to $2^r$.

How to Handle Keys of Type String

- Just treat a string as a long binary number.

  Example: Suppose the key is the string 'KEY'
  In ASCII, this is 1001011 1000101 1011001.
  Treat this as a binary number and use previous hashing functions.

- Folding. Suppose we are given a key comprising 11 letters $a_1a_2a_3a_4a_5a_6a_7a_8a_9a_{10}a_{11}$.

\[
\begin{array}{cccc}
  a_1 & a_2 & a_3 & a_4 \\
+ & a_5 & a_6 & a_7 & a_8 \\
+ & a_9 & a_{10} & a_{11} \\
\end{array}
\]
Collision Resolution Method I: Chaining

For chaining, we put entries hashed to the same index into a linked list.

The advantages of chaining are:
- It is easy to locate a key when there are collisions.
- The hash table needs only space for pointers.
- Deletion of an item is straight-forward.

The disadvantage of chaining is that we need additional space for pointers along with the data.

Example: Suppose \( h(key) = key \% 100 \)

Keys: 4203 3606 1200 1302 8802 4106
Collision Resolution Method II: 
Open Addressing

- If $h(key) = h_0$ and is occupied, then we try, or “probe,” positions $h_1, h_2, h_3 \ldots$, until we find an opening slot.

- Common probing techniques: linear, quadratic, random, double hashing

---

Linear Probing

- $h_i = (h_0 + i) \%$ CAPACITY

- That is, we try $h_0 + 1, h_0 + 2, h_0 + 3, \ldots$ until an empty slot is found.

- the “$\%$ CAPACITY” part ensures “wrapping around” after unsuccessfully probing the last location in the table.

When the table becomes about half full, linear probing has a tendency toward clustering; that is, data occupy long strings of adjacent positions with gaps between the strings.
Keys: 211, 143, 160, 821, 223, 221

\[ M = 10 \]

Quadratic Probing

- \[ h_i = (h_0 + i^2) \mod \text{CAPACITY} \]
  
  That is, we try \( h_0 + 1, h_0 + 4, h_0 + 9, \ldots \) until an empty slot is found.

- Typically, quadratic probing will not probe all locations.

- For prime \text{CAPACITY} values, \((\text{CAPACITY}+1)/2\) probes will bring you back to the starting point; when that happens, there is no point in going further. (See pages 402 to 403 of Kruise for a proof.)
Example

CAPACITY=7, key = 702

\[ h_0 = 2 \]
\[ h_1 = (h_0 + 1^2) \mod 7 = 3 \]
\[ h_2 = (h_0 + 2^2) \mod 7 = 6 \]
\[ h_3 = (h_0 + 3^2) \mod 7 = 4 \]
\[ h_4 = (h_0 + 4^2) \mod 7 = 4 \]
\[ h_5 = (h_0 + 5^2) \mod 7 = 6 \]
\[ h_6 = (h_0 + 6^2) \mod 7 = 3 \]
\[ h_7 = (h_0 + 7^2) \mod 7 = 2 \]

We have completed a cycle, and half of the table entries are not probed.

- consider inserting 2, 72, 702, 7002, and 70002 to an empty, size-7 hash table that uses quadratic probing

- for all these keys, \( h_0 = 2, h_1 = 3, h_2 = 6, \) and \( h_3 = 4. \)

- This is typically acceptable with hash tables, for “probing half of the table” most probably indicates flaws in system design.

☞ a hash table is typically large to minimize the chances of collisions.
Double Hashing

- For double hashing, we use a second hashing function $h'(key)$.
- If location $h(key) = h_0$ is occupied, we probe

$$h_0 + h'(key), h_0 + 2 \cdot h'(key), h_0 + 3 \cdot h'(key), \ldots$$

until an empty space is found.

Rehashing

- For rehashing, we also use two hashing functions, $h$ and $h'$.
- If $h(key) = h_0$ is occupied,
  then we try $h_1 = h'(h_0)$,
  then we try $h_2 = h'(h_1)$,
  then we try $h_3 = h'(h_2)$, and so forth.
- A good choice of $h$ and $h'$ is as follows:
  - $h(key) = key \ % \ CAPACITY$
  - $h'(x) = 1 + (x \ % \ (CAPACITY - 2))$
  - Both CAPACITY and CAPACITY – 2 should be prime (for example, 811 and 809).
Implementing Open-Address Hashing with Linear Probing

template <class Entry>
class Table
{
public:
    static const int CAPACITY = 811;
    Table(); ~Table();

    void insert (const Entry& entry);
    void remove (int key);
    bool is_present (int key) const;
    void find (int key, bool& found, Entry& result) const;
    int size() const {return used;}

private:
    static const int NEVER_USED = -1;
    static const int PREVIOUSLY_USED = -2;

    Entry data[CAPACITY];
    int used;
    int hash (int key) const;
    void find (int key, bool& found, int& index) const;
}
template <class Entry>
void Table<Entry>::find (int key, bool& found, int& i) const
{
    int count = 0;
    int i = hash (key);
    while (count < CAPACITY && data[i].key != key &&
        data[i].key != NEVER_USED) {
        count++;
        i = (i+1) % CAPACITY;
    }
    found = data[i].key == key;
}

template <class Entry>
void Table<Entry>::insert (const Entry& entry) const
{
    bool already_present;
    int i;
    find (entry.key, already_present, i);

    if (!already_present) {
        i = hash(entry.key);
        while (data[i].key != NEVER_USED ||
            data[i].key != PREVIOUSLY_USED)
            i = (i+1) % CAPACITY;
        used++;
        data[i] = entry;
    }
}
template <class Entry>
void Table<Entry>::remove (int key)
{
    bool present;
    int i;
    find (key, present, i);
    if (present) {
        data[i].key = PREVIOUSLY_USED;
        used--;
    }
}

Performance

- load factor
\[
\alpha = \frac{\text{# of occupied table entries}}{\text{CAPACITY}}
\]

- In open-address hashing with linear probing, a nonfull hash table, and no deletions, the average number of locations examined in a successful search is approximately:
\[
\frac{1}{2} \left( 1 + \frac{1}{1 - \alpha} \right)
\]

Given a table with CAPACITY=811 and 611 entries occupied, we have \( \alpha \approx 0.8 \). We expect successful searches to examine
\[
\frac{1}{2} \left( 1 + \frac{1}{1 - 0.8} \right) = 3.0 \text{ slots.} \]
In open-address hashing with double hashing, a nonfull hash table, and no deletions, the average number of table entries examined in a successful search is approximately:

\[-\frac{\ln(1 - \alpha)}{\alpha}\]

Given a table with CAPACITY=811 and 611 entries occupied, we expect successful searches to examine \(-\frac{\ln(1-0.8)}{0.8}\) = 2 slots.

In chained hashing, the average number of table entries examined in a successful search is approximately:

\[1 + \frac{\alpha}{2}\]

Note that, with chaining the table can be overloaded (that is, \(\alpha \geq 1\)).

### An Empirical Comparison

<table>
<thead>
<tr>
<th>Load factor</th>
<th>0.1</th>
<th>0.5</th>
<th>0.8</th>
<th>0.9</th>
<th>0.99</th>
<th>2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Successful search, average # of probes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chaining</td>
<td>1.04</td>
<td>1.2</td>
<td>1.4</td>
<td>1.4</td>
<td>1.5</td>
<td>2.0</td>
</tr>
<tr>
<td>Open, quadratic probes</td>
<td>1.04</td>
<td>1.5</td>
<td>2.1</td>
<td>2.7</td>
<td>5.2</td>
<td>NA</td>
</tr>
<tr>
<td>Open, linear probes</td>
<td>1.05</td>
<td>1.6</td>
<td>3.4</td>
<td>6.2</td>
<td>21.3</td>
<td>NA</td>
</tr>
<tr>
<td>Unsuccessful search, average # of probes</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chaining</td>
<td>1.1</td>
<td>1.5</td>
<td>1.8</td>
<td>1.9</td>
<td>1.99</td>
<td>2.0</td>
</tr>
<tr>
<td>Open, quadratic probes</td>
<td>1.13</td>
<td>2.2</td>
<td>5.2</td>
<td>11.9</td>
<td>126</td>
<td>NA</td>
</tr>
<tr>
<td>Open, linear probes</td>
<td>1.13</td>
<td>2.7</td>
<td>15.4</td>
<td>59.8</td>
<td>430</td>
<td>NA</td>
</tr>
</tbody>
</table>