Resolving collisions

Several approaches
1. **Chain hashing** - all keys mapped to the same cell are stored in a linked list. (Less popular in practice - dynamic memory allocation is slow, multiple vulnerabilities, less friendly to compiler-optimization, GPU unfriendly…
2. **Cuckoo hashing** - will discuss later
3. Resolving collisions by **open addressing** - most popular.

Resolving collisions by open addressing

No storage is used outside of the hash table itself.

Each cell could contain at most one key.

The same key $k$ might be mapped by $h(k)$ to different locations in the table, depending on availability.

When either searching $k$ or searching for a place for $k$, we will check

The **first** index that we search $k$. If fail
The **second** index that we search $k$. If fail
The **third** index that we search $k$. If fail etc

When should we give up? (will see in next slides)

How should we find these indexes?

$h(k, i)$- a hash function that takes two parameters:
- Key $k$
- Trial number $i$ (first trail has index 0)

Inserting a key $k$:
we check $T[h(k,0)]$. If empty we insert $k$, there. Otherwise, we check $T[h(k,1)]$. If empty we insert $k$, there. Otherwise,… otherwise etc for $h(k,2), h(k,3), …, h(k,m-1)$.

Finding a key $k$:
we check whether $T[h(k,0)] == k$. If not, if empty, stop. otherwise we check whether $T[h(k,1)] == k$. If not, if empty, stop. otherwise etc for $h(k,2), h(k,3), …, h(k,m-1)$.

Example of Insertion

Hash function: $h(k, i) = (k+i) \mod 8$

$k$-key: $i$ is the attempt number (start at 0)

- insert(12). $h(12,0)=4$
  Read: The first attempt ($i=0$) checks $T[h(12,0)]$. It is free
- insert(15). $h(15,0)=7$
- insert(20). $h(20,0)=4$ (collision) $h(20,1)=(20+1)\mod 8=5$
- insert(23). $h(23,0)=7$ (collision) $h(23,1)=0$
- insert(28). $h(28,0)=4$ (collision) $h(28,1)=5$ (collision); $h(28,2)=6$
Searching a key. Example on the same table
Hash function: \( h(k,i) = (k+i) \mod 8 \)

Finding a key \( k \):
we check if \( T[h(k,0)] = k \). If not, if empty, stop. otherwise
we check if \( T[h(k,1)] = k \). If not, if empty, stop. other etc

```
<table>
<thead>
<tr>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>28</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>
```

\( k \)-key. \( i \) is the attempt number (start at 0)

'Search' uses the same probing sequence. The Search stops once it hits an empty cell, or \( i=n-1 \).


Search(16). \( h(16,0)=0 \). \( T[0] \neq 16 \). Next check \( h(16,1)=5 \), but \( T[5] \)-empty. Search terminates - 16 not in table.

Maintenance

Scan the table from time to time, and get rid of all of all dummies. Re-insert each key, If the table needs to be expanded - good opportunity to use the dynamic table technique and re-hash.

Searching a key. Example on the same table
Hash function: \( h(k,i) = (k+i) \mod 8 \)

```
<table>
<thead>
<tr>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>28</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>15</td>
</tr>
</tbody>
</table>
```

\( k \)-key. \( i \) is the attempt number (start at 0)

- Next, delete 20, and then lets again search 28.
- The search wrongly stops at the empty cell that used to contain 28. Error
- Solution: Place a dummy to indicate that this cell used to contain a key, but this key was deleted. The 'search' treats this cell as 'nonempty' and continues the probing sequence. The search stops only when reaching a cell that is "really" empty.
- When inserting a new key, we can replace the dummy with a real key. Example - inserting 13 will override the dummy

'Search' uses the same probing sequence. The Search stops once it hits an empty cell, or \( i=n-1 \).


Search(16). \( h(16,0)=0 \). \( T[0] \neq 16 \). Next check \( h(16,1)=5 \), but \( T[5] \)-empty. Search terminates - 16 not in table.

Probing strategies

**Linear probing:**
Given an ordinary hash function \( h'(k) \), linear probing uses the hash function
\[ h(k,i) = (h'(k) + i) \mod m. \]
This method, though simple, suffers from *primary clustering*, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.

Theoretically, inferior method.

In practice, is the fastest method. Why? In the memory hierarchy, locality is a winner. If we accessed \( T[i] \), then it is likely that \( T[i+1] \) is awaiting in cache.
**Probing strategies**

**Double hashing**

Given two ordinary hash functions \( h_1(k) \) and \( h_2(k) \), double hashing uses the hash function

\[ h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m. \]

This method generally produces excellent results, but \( h_2(k) \) must be relatively prime to \( m \). One way is to make \( m \) a power of 2 and design \( h_2(k) \) to produce only odd numbers.

**Analysis of open addressing**

**Theorem.** If the data is distributed well enough (detailed dropped), the expected number of probs for insert/delete/find is \( \geq 1/(1 - \alpha) \)

\[ \alpha = \frac{\text{number of keys}}{\text{number of cells}} \]

Example: \( \alpha = 0.99 \). Need **100** probs on average.

Example: \( \alpha = 0.5 \). Need **2** probs on average.

Conclusion: Keep \( m \geq 2n \). Use dynamic arrays if needed.

\[ \sum_{i=2}^{\infty} \frac{i}{\alpha^i} = \frac{(2\alpha - 1)}{((\alpha - 1)^2 \alpha)} \]

The expected number of probs, until an empty slot is found

Recall \( \alpha = n/m \) - load factor.

Assumption: At every \( i \), the probability of hitting cell \( j \) is \( 1/m \) (uniformly).

Let's call a prob a "success" if we hit an empty cell, and "fail" if hit an occupied cell.

The sequence probs ends with a successful prob.

The probability that exactly 0 fail probs are needed is \( 1 - \alpha \)

(success on first try)

The probability that exactly 1 fail probs is needed is \( \alpha(1-\alpha) \)

(fail, then success)

The probability that exactly 2 fail probs are needed is \( \alpha^2(1-\alpha) \)

(fail, fail then success)

The probability that exactly 3 fail probs are needed is \( \alpha^3(1-\alpha) \)

(fail, fail, fail then success)

The probability that exactly \( j \) fail probs are needed is \( \alpha^j(1-\alpha) \)

(\( j \) fails, then success)

So the expected number of probs is

\[ 1_{\text{successful prob}} + (1 - \alpha) \sum_{j=1}^{\infty} \alpha^j \cdot j_{\text{fails}} \]

Expected number of probs (cont)

Conclusions:

If \( \alpha = 0.5 \), the expected number of probs is **2**

If \( \alpha = 0.95 \), the expected number of probs is **300**

If \( \alpha = 0.98 \), the expected number of probs is **10^4**

Conclusion: Keep \( m \geq 2n \).

Rehash twice a day: first thing every morning, and before bed time.
Conclusions:
If $\alpha = 0.5$, the expected number of probs is 2
If $\alpha = 0.95$, the expected number of probs is 300
If $\alpha = 0.98$, the expected number of probs is $10^4$

Conclusion: Keep $m \geq 2n$.

Rehash twice a day: first thing every morning, and before bed time.