

**Hashing (just the basics)**

Thanks to
Prof. Charles E. Leiserson

---

**Symbol-table problem**

Symbol table $T$ holding $n$ records:

- **key[x]**
- Other *fields* containing *satellite data*

Operations on $T$:
- $\text{INSERT}(T, x)$
- $\text{DELETE}(T, x)$
- $\text{SEARCH}(T, k)$
Symbol-table problem

Symbol table $T$ holding $n$ records:

- Operations on $T$:
  - INSERT($T, x$)
  - DELETE($T, x$)
  - SEARCH($T, k$)

How should the data structure $T$ be organized?

Hash tables and hash functions

We always have a table (cubby). Each cell has an index. The index is a number between $0..m-1$.

A hash function $h$ computes for every possible key an index in a hash table $\{0, 1, \ldots, m-1\}$.

As each key is inserted, $h$ maps it to a slot of $T$. 
As each key is inserted, \( h \) maps it to a slot of \( T \).
As each key is inserted, \( h \) maps it to a slot of \( T \).

When a record to be inserted maps to an already occupied slot in \( T \), a collision occurs.
Resolving collisions by chaining

- Records in the same slot are linked into a list.

\[ h(49) = h(86) = h(52) = i \]

Analysis of chaining

Let \( n \) be the number of keys in the table, and let \( m \) be the number of slots.

Define the load factor of \( T \) to be:

\[ \alpha = \frac{n}{m} \]

- \( \alpha \) is the average number of keys per slot.

We will try to keep the this value no larger than 1 (same number of keys and slots).

Search cost

Expected time to search for a record with a given key is \( \Theta(1 + \alpha) \).

- Apply hash function and access slot
- Search the list
Search cost

Expected time to search for a record with a given key = $\Theta(1 + \alpha)$.

- apply hash function and access slot
- search the list

Expected search time = $\Theta(1)$ if $\alpha = O(1)$, or equivalently, if $n = O(m)$.

What to do if table too dense

Once $\alpha$ is too large

It does not affect correctness, but affects performance.

Once we have a chance, re-double the table (and compute a new hash function)

Example (from GeeksforGeeks)

Start with a table of a small size
If it does not store the pointers to the linked list
Then re-double the size since 2,4,8,

Total time: if it takes $O(1)$ to re-insert a key, the total time for inserting $n$ keys is $n \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots\right) = 2n$

Choosing a hash function

Desirata:

- A good hash function should distribute the keys uniformly into the slots of the table.
- Regularity in the key distribution should not affect this uniformity.
- Hope: if $k_1 \neq k_2$ in any bit, then there is a good chance $h(k_1) \neq h(k_2)$
- Functions that ignore some bits (e.g. $h(k) = k \mod 100$, $h(k) = k \mod 1024$) should be used only if know enough about the data distribution to think that this is not an issue.
**Division method**
Assume all keys are integers, and define
\[ h(k) = k \mod m. \]

**Deficiency:** Don’t pick an \( m \) that has a small divisor \( d \). A preponderance of keys that are congruent modulo \( d \) can adversely affect uniformity.

**Extreme deficiency:** If \( m = 2^r \), then the hash doesn’t even depend on all the bits of \( k \):
- If \( k = 10110001111011010 \) and \( r = 6 \), then
  \[ h(k) = 0110101000111011010 \]
**Division method (continued)**

\[ h(k) = k \mod m. \]

Pick \( m \) to be a prime not too close to a power of 2 or 10 and not otherwise used prominently in the computing environment.

**Annoyance:**
- Sometimes, making the table size a prime is inconvenient.
- But, this method is popular, although the next method we’ll see is usually superior.

---

**Multiplication method**

Assume that all keys are integers. Pick a constant integer \( A \), and set

\[ h(k) = (Ak) \mod m \]

\( A \) is an odd integer

Other variant of the multiplication method:
- Pick \( A \) as a non-integer number
- \( A=2.71828182846, \) or \( A=\sqrt{2} = 1.41421356237 \)

\[ h(k) = \left[ m \left( Ak - \lfloor Ak \rfloor \right) \right] \]

Note - the part in the red parenthesis is a float in \((0,1)\).
Multiply my \( m \) gives a float in \((0,m)\). The second floor just makes it a legit index in the table \([0…m-1]\).

---

**Multiplication method example**

Variant 3: \( A \) is a large integer, but the value of \( h(k) \) is the number that several digits in the ‘middle’ of the \((Ak)\).

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & \times A \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & = k \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\
\hline
& & & & & & & & & & & & & h(k)
\end{array}
\]
Let's understand why all the variants of the multiplication method works nicely

- Think about a series of keys
- $k_1 = 1$, $k_2 = 2$, $k_3 = 3$...
- We hope that $h(k_1), h(k_2), h(k_3)$... fall in different and pairwise remote locations in the table.

\[
\begin{array}{c}
1 & 0 & 1 & 1 & 0 & 0 & 1 \\
1 & 1 & 0 & 1 & 0 & 1 & 1
\end{array}
\]

- $A = k_h(k)$

**Modular wheel**
**Multiple hash functions:**

Applications to distributed database: We need to store a large number $n$ of records.

1. Let's think about a system with 8 disks.
2. Or 8 users, each in a different locations.

- RAID of disks is a device that contains multiple disks (sometimes sharing parts).
- Each individual disk is prone to failures. Want to maximize robustness and load fairness.
  - Robustness to disk failures: We should still be able to access all our data, even if two disks crashed. So each record needs to be stored on multiple disks.
  - Load fairness: Each disk should store a small portion of the database, and these portions should be split fairly. So each disk should contain approximately $\frac{3n}{8}$ records.
- Efficiency: Search time should be small. When searching for a key, should not have to check each individual disk.
- Need to support: `Insert(k)/delete(k)/find(k)`.
- We don't know the data in advance—changing dynamically.
- So even as new records appear, need to decide which 3 disks will store it. (Once a query finds it, need to be able to find those 3 disks.)

---

**Let's understand why all the variants of the multiplication method works nicely**

- Think about a series of keys
- $k_1 = 1, k_2 = 2, k_3 = 3, \ldots$
- We hope that $h(k_1), h(k_2), h(k_3), \ldots$ fall in different and pairwise remote locations in the table.

\[
\begin{array}{c}
1011001 = A \\
1101011 = k \\
\hline
h(k) = A \\
2A
\end{array}
\]

**Modular wheel**

---

**Let's understand why all the variants of the multiplication method works nicely**

- Think about a series of keys
- $k_1 = 1, k_2 = 2, k_3 = 3, \ldots$
- We hope that $h(k_1), h(k_2), h(k_3), \ldots$ fall in different and pairwise remote locations in the table.

\[
\begin{array}{c}
1011001 = A \\
1101011 = k \\
\hline
h(k) = A \\
3A
\end{array}
\]

**Modular wheel**
Multiple hash functions:

It is convenient sometimes to have multiple hash functions \( h_1(k), h_2(k), h_3(k) \)

We can generate them by picking 3 constants \( A_1, A_2, A_3 \) for example

\[
(A_1 \mod 8, (A_2 \mod 8, (A_3 \mod 8)
\]

We don’t discuss here where in the disk each record is stored—orthogonal discussion.

- We don’t know the data in advance—changing dynamically.
- Insert(\( k \)). A new record with key \( k \) appeared. Compute Insert(\( k \)) into disk whose index is \( h_1(k) \), \( h_2(k), h_3(k) \).

Similarly insert copies of \( k \) into disks \( h_1(k), h_2(k), h_3(k) \) and check these disks. If don’t find, it is either because was never inserted or due to disk failures.

\[ h_1(k) = (3k \mod 8), (5k \mod 8), (7k \mod 8) \]

Dot-product method. Hashing large files.

In many applications, the key is too long to be considered a single number.

E.g. \( k = "BDCZ" \).

In general, we need a hash functions that could be used on very long keys, as text documents, books, images, DNA, geometric structures, malware, viruses...

Expressing the key as a single number is not useful.

Idea: Remember that if the key \( k \) is a small number, we could use the multiplication method, and set \( h(k) = (Ak) \mod m \)

Now if the key is very large, lets break it into several small pieces, so instead of tread the key as a single number, lets think about it as a vector (or a list) consisting of several numbers.

Instead of a single constant \( A \), we compute multiple and different constants \( a_1, a_2, a_3, a_4 \)

We decompose the key into characters, multiply each by a different constant and sum (modulo \( m \)).

Example:

\[ h(BDCZ) = (a_1 \cdot 66 + a_2 \cdot 68 + a_3 \cdot 67 + a_4 \cdot 99) \mod m \]

(\( m \) is the size of the hash table. The max value of \( 'B' \) is 66 and of \( 'Z' \) is 99).

- Computing all constants \( a_i \) is very simple. Pick random integers between \( 1 \) and \( m-1 \).
- Excellent in practice, and in theory.
- Involve one pass of the file, in the case of very long keys.

Dot product method-cont.

Before any data item arrives, decide about the size \( m \) of the hash table.

\( m \) should be prime, and \( >2n \).

Let \( m \approx 2^n = 1M \)

Pick at random constants \( \overrightarrow{a} = (a_0, a_2, \ldots a_r) \).

Each \( a_i \) is picked individually at random uniformly \( 1 < a_i < m - 1 \)

Now the first key \( k \) arrive. Lets break it into pairs of characters \( k^{\text{char}} \) according to section 1223(b) a nonprofit organization...

Break into pairs of characters, and for each pair, compute its numeric value using base 256 (ASCII).

\[
k = \text{According to sect} \overrightarrow{a} \cdot k^{\text{char}}
\]

\[
\overrightarrow{a} \cdot k^{\text{char}} = a_0 \cdot k_0 + a_2 \cdot k_2 + a_4 \cdot k_4...
\]

Where \( k_0 = 'A' \cdot 256 + 'c' = 65 \cdot 256 + 99. \)

Finally \( h_{\overrightarrow{a}}(k) = \left( \sum_{i=0}^{r} a_i \cdot k_i \right) \mod m \)
A deeper look at the dot product method

- Obviously, our aim is to minimize collisions
- From now on, assume $m$ (the table size) is a prime number.
- Assume all our keys $K = \{k_1, k_2, \ldots, k_n\}$ are numbers, between $0..m-1$.
- No key appears twice.
- Pick any constant integer $\alpha \in [1..m-1]$. Lets consider the hash function $h(x) = (\alpha x) \mod m$.

**Lemma 1**: for every $Y \in [0..m-1]$, there is a unique $t \in [0..m-1]$ such that $(at) \mod m = Y$.

**Lemma 2**: good news: The lemma guarantees that the hash function $h(x) = (\alpha x) \mod m$ will map the keys of $K$ to different cells of the hash table. No collisions at all.

**Bad news**: this guarantee is waved if we don’t require that all keys of $K$ are $< m$. For example, lets play with $h(x) = (3x) \mod 5$. Then $h(3) = h(8)$. So by itself, this is not very helpful. We will see next how to use it more efficiently.

**Before continuing, let’s rewrite Lemma 1**:

**Lemma 3**: for every fixed $Y \in [0..m-1]$, and every fixed $x_i \in [1..m-1]$, there is exactly one value $\alpha_i \in [0..m-1]$ such that $(\alpha_i x_i) \mod m = Y$.

More on dot-product method

- New think about a set of keys $K = \{k_1, k_2, \ldots\}$, where each key is a point $p = (x, y)$ (the every). These are points that we need to store in a hash table.
- Lets pick the table size $m = 12$, and a prime example $p = 30$, so we pick $m = 13$.
- We want to choose a hash function that would map these points to the hash table.
- If we know which keys are in $K$, we could create a perfect hash function that would create no collisions. But usually, we don’t choose $K$, and even if we do, it does not worth the trouble.

**Also**: Pick at random two constants $\alpha$, $\beta$ in the range $1..m$. When we need to decide at which cell to store the point $p = (x, y)$, we use hash function $h(p) = (\alpha x + \beta y) \mod m$.

**Lemma 4**: the probability that $h(p_1) = h(p_2)$ is $\leq \frac{1}{m}$. Thats, for any two points, the probability of a collision is really small.

**Proof**: assume $\alpha_1, \alpha_2, \beta_1, \beta_2$ are fixed. Since they are not the same point, assume $\alpha_1 \neq \alpha_2$ (the case $\alpha_1 = \alpha_2$ is symmetrical).

If $\alpha_1 \neq \alpha_2$ then $(\alpha_1 x + \beta_1 y) \mod m \neq (\alpha_2 x + \beta_2 y) \mod m$, which implies $\alpha_1 x + \beta_1 y \neq \alpha_2 x + \beta_2 y$ and $h(p_1) \neq h(p_2)$.

- Think about this step. The values of $\alpha_1, \beta_1, \alpha_2, \beta_2$ are fixed, and we have no control about them. We just picked $\alpha, \beta$, so the value of $h(p) = (\alpha x + \beta y) \mod m$ is fixed. The value is $\mod m$ in other words. Here we do a mental experiment: check the case $x = 0, 1, 2, \ldots, m-1$.

In practice, instead of checking these values directly, we just pick $\alpha, \beta$ at random.

**Dot-product method - cont**

For my pick $\alpha, \beta$ at random from the range (1..m). Each pair $(\alpha, \beta)$ is a unique point.

**Conclusion from Lemma 3**: the probability that $h(p_1) = h(p_2)$ (that is, a collision occurs) is $\leq \frac{1}{m}$.

Now consider a set $K = \{k_1, k_2, \ldots\}$ of $n$ keys. Let us think about the expected number of collisions between $p_1$ and the other keys of $K$. Using the same ideas that we used for the height of SkipList Analysis, the number is smaller than the sum of each individual probability. That is:

$$\sum_{x \neq x_1} \frac{1}{m} \leq \frac{\binom{m}{2}}{m} = \frac{m(m-1)}{2m} = \frac{m-1}{2}$$

Thus, assume that $K$ is a set of $k$ keys ($K = \{k_1, k_2, \ldots\}$, each is a number in $[0..m]$). Which hash function should we use?

- At random, we pick $\alpha, \beta$ at random: $\alpha \neq \beta$.

**Attempt 1**: Pick $\alpha = 0 \mod m$ (for simplicity, it’s very well, but it requires guarantees. A vicious and greedy could pick the keys of $K$ which are bad for almost every chance of $\alpha$.

**Better approach**: For every key $k_i$, compute $E_i = \text{hash}(k_i)$. We are back to the case of $x$ keys.

**Example for $\alpha = 10, 11, 12$**. Then $E_i = \text{hash}(k_i)$ are in $[1..m]$. We will use the first 2 bytes for $\alpha$ and the last two for $\gamma$.

**Another example**: $\alpha = 11, \beta = 35$. Then $E_i = \text{hash}(k_i)$ are in $[1..m]$.

**Instead of expressing $E_i$ in base $m$, we could use any other way to express $k_i$ as two numbers $(s, g)$, both $\leq m-1$. For example, if $m \leq 25$, and $\alpha_i < 25$ then $\gamma_i = 0$. We will use the first 2 bytes for $s_i$ and the last two for $g_i$.**

**Similarly**, if each $k_i$ is a number between 0 and 100, we will pick at random 3 values $\alpha, \beta, \gamma \in [0..m-1]$. We express each $k_i$ using 7 digits $s_0 \ldots s_6$. All in $[0..m-1]$. So $\alpha_i = (m^6 + s_0 + \gamma_i) \mod m$.

**If the length of the key is unlimited (e.g. documents)**, we use round robin.
Universal family of hash functions

- We have a set of hash functions \( H = \{ h_1(k), \ldots, h_L(k) \} \).

- We say that it is universal iff: for every two keys \( k, k' \in U \), if we pick at random \( h(k) \in H \), then the probability of a collision \( h(k) = h(k') \) is \( \leq 1/m \).

- That is, only \( \frac{1}{m} \) of the functions of \( H \) cause collisions of \( k, k' \).

- If we think about all the possible hash functions \( h(k, y) = (ax_i + \beta y_i) \mod m \).

- When we change \( a, \beta \) (both in \([0..m-1]\) ), we create different members of the family.

- We just saw that this family is universal.

- It guarantees that the probability of collusion between \( h(k, y) \) is \( \frac{1}{m} \), and that the average number of collisions between \( k \) and any other member of \( K \) is \( \leq 1/2 \).

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 trial has index 0)

Resolving collisions by open addressing

- No storage is used outside of the hash table itself.

- The hash function depends on both the key and probe number:

- \( h(k, i) \):
  - input is a pair: a key and a trial number \([0, 1, 2, \ldots, m-1]\)
  - Output: Always a legit index in the table \( T[\cdot] \): a number in the range \([0, 1, \ldots, m-1]\)

- E.g. \( h(k, i) = (k + i) \mod m \)
  - \( h(k, i) = (k + i \cdot \beta) \mod m \)
  - Here \( h \) is some other hash function
  - \( f(k, i) = (k + i' \beta) \mod m \)

- 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, we check \( T[h(k, 2)] \), \( T[h(k, 3)] \), \ldots, \( T[h(k, m-1)] \).

- Finding a key \( k' \):
  - we check whether \( T[h(k', 0)] \) \( \rightarrow k' \).
  - If not, if empty, stop. otherwise we check whether \( T[h(k', 1)] \) \( \rightarrow k' \).
  - If not, if empty, stop. otherwise we check whether \( T[h(k', 2)] \), \( T[h(k', 3)] \), \ldots, \( T[h(k', m-1)] \).
Example of Insertion
Hash function: \( h(k,i) = (k+i) \mod 8 \)
k - key, \( i \) is the attempt number (start at 0)

\[
T
\]

• insert(12). \( h(12,0) = 4 \)

Read: The first attempt (\( i=0 \)) checks \( T[h(12,0)] \). It is free
Example of Insertion
Hash function: \( h(k,i) = (k + i) \mod 8 \)

- **insert(12).** \( h(12,0) = 4 \)
  - Read: The first attempt \((i=0)\) checks \(T[h(12,0)]\). It is free

- **insert(15).** \( h(15,0) = 7 \)
  - Read: The first attempt \((i=0)\) checks \(T[h(15,0)]\). It is free

- **insert(12).** \( h(12,0) = 4 \)
  - Read: The first attempt \((i=0)\) checks \(T[h(12,0)]\). It is free
  - **insert(15).** \( h(15,0) = 7 \)
  - Read: The first attempt \((i=0)\) checks \(T[h(15,0)]\). It is free

---

Example of Insertion
Hash function: \( h(k,i) = (k + i) \mod 8 \)

- **insert(12).** \( h(12,0) = 4 \)
  - Read: The first attempt \((i=0)\) checks \(T[h(12,0)]\). It is free

- **insert(15).** \( h(15,0) = 7 \)
  - Read: The first attempt \((i=0)\) checks \(T[h(15,0)]\). It is free

---

Example of Insertion
Hash function: \( h(k,i) = (k + i) \mod 8 \)

- **insert(12).** \( h(12,0) = 4 \)
  - Read: The first attempt \((i=0)\) checks \(T[h(12,0)]\). It is free

- **insert(15).** \( h(15,0) = 7 \)
  - Read: The first attempt \((i=0)\) checks \(T[h(15,0)]\). It is free
Example of Insertion
Hash function: $h(k,i) = (k+i) \mod 8$

- $k$ is the 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)
  - $h(20,1) = (20+1) \mod 8 = 5$
Example of Insertion

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

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

- Insert(4). \( h(12,0) = 4 \)
  Read: The first attempt (i=0) checks \( T[h(12,0)] \). It is free

- Insert(5). \( h(15,0) = 7 \)

- Insert(20). \( h(20,0) = 4 \) (collision)
  \( h(20,1) = (20 + 1) \mod 8 = 5 \)
  \( T[5] \) is empty. Place 20 at \( T[5] \)

- Insert(23). \( h(23,0) = 7 \) (collision)
  \( h(23,1) = 0 \)

- Insert(28). \( h(28,0) = 4 \) (collision)
  \( h(28,1) = 5 \) (collision)
  \( h(28,2) = 6 \)
Example of Insertion

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

- \( k \)-key. \( i \) is the attempt number (start at 0)
- \( \text{insert}(12). h(12,0)=4 \)
  - Read: The first attempt \((i=0)\) checks \( T[4] \). It is free
  - \( \text{insert}(15). h(15,0)=7 \)
  - \( \text{insert}(20). h(20,0)=4 \) (collision)
    - \( h(20,1)=(20+1)\mod 8=5 \)
    - \( T[5] \) is empty. Place 20 at \( T[5] \)
  - \( \text{insert}(23). h(23,0)=7 \) (collision)
    - \( h(23,1)=0 \)
  - \( \text{insert}(28). h(28,0)=4 \) (collision)
    - \( h(28,1)=5 \) (collision)
    - \( h(28,2)=6 \)

Searching a key. Example on the same table

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

- \( k \)-key. \( i \) is the attempt number (start at 0)
- \( \text{Search} \) uses the same probing sequence. The Search stops once it hits an empty cell, or \( i=n-1 \).
- Search(16). \( h(16,0)=0 \), \( T[0]=16 \). Next check \( h(16,1)=5 \), but \( T[5] \) is empty. Search terminates - 16 not in table.

Finding a key \( k \): we check if \( T[h(k,0)] \neq k \). If not, if empty, stop otherwise we check if \( T[h(k,1)] \neq k \). If not, if empty, stop otherwise...
Searching a key. Example on the same table

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

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

Table:

<table>
<thead>
<tr>
<th>T</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>15</td>
<td>28</td>
<td>23</td>
</tr>
</tbody>
</table>


Next, delete 28.

Next, delete 28.
Searching a key. Example on the same table

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

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

Next, delete 28.
Now, lets search 28 again.

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


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

The search wrongly stops at the empty cell that used to contain 28. Error.
Searching a key. Example on the same table

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

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

<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>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>23</td>
</tr>
</tbody>
</table>

Next, delete 28.
Now, lets search 28 again.
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 the key was deleted. The search skips this cell as ‘nonempty’ and continues the probing sequence. The search stops only when reaching a cell that is ‘empty’ or empty.
Searching a key. Example on the same table

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

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

Next, delete 20.

Now, let's search 28 again.

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.

The search uses the same probing sequence. The search stops once it hits an empty cell, or $i = n - 1$.


When inserting a new key, we can replace the dummy with a real key. Example - inserting 13 will override the dummy.

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.
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.

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

\[
\alpha = \frac{\text{number of keys}}{\text{number of cells}}
\]
**Analysis of open addressing**

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

$$\alpha = \frac{\text{number of keys}}{\text{number of cells}}$$

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.