Hashing (just the basics)

Thanks to
Prof. Charles E. Leiserson
Symbol-table problem

Symbol table $T$ holding $n$ records:

$record$

$x$ -> $key[x]$

Other *fields* containing *satellite data*
Symbol-table problem

Symbol table $T$ holding $n$ records:

- Key $[x]$
- Other fields containing satellite data

Operations on $T$:
- **INSERT**($T$, $x$)
- **DELETE**($T$, $x$)
- **SEARCH**($T$, $k$)
Symbol-table problem

Symbol table $T$ holding $n$ records:

- $x$ is the key
- Other fields containing satellite data

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 \( \{0, 1, \ldots, m-1\} \):

\[
T \quad 0 \\
\vdots \\
m-1
\]
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, ..., $m-1$\}:

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

\[ T \]

\[ 0 \]

\[ 3 \]

\[ 7 \]
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 $T$. 

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

$$h(k_1), h(k_2)$$
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, ..., m-1\}:

As each key is inserted, $h$ maps it to a slot of $T$. 
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 $T$. 
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, ..., m-1\}:

As each key is inserted, $h$ maps it to a slot of $T$. 
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 $\{0, 1, \ldots, m-1\}$:

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

$$\alpha = \frac{n}{m}$$

= 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 $= \Theta(1 + \alpha)$.
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 effect corrections, but effects performances.
Once we have a chance, re-double the table (and compute a new hash function)

Example (credit GeeksforGeeks)

Start with a table of a small size
(Fig does not show the pointers to the linked list
When table too dense, double its size
sizes: 2,4,8,....

Total time: if it takes $O(1)$ to re-insert a key, the total time for inserting $n$ keys is $n\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} \ldots \right) \leq 2n$
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.
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 \( k = 1011000111011010_2 \) and \( r = 6 \), then \( h(k) = 011010_2 \).
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 \( k = 1011000111011010_2 \) and \( r = 6 \), then
  \[ h(k) = 011010_2. \]
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

$$h(k) = (A \cdot k) \mod m$$

$A$ is an odd integer

Other variant of the multiplication method:

Pick $A$ as a non-integer number

$A = 2.71828182846$, or $A = \sqrt{2} = 1.41421356237$

$$h(k) = \left\lfloor m \left( (A \cdot k) - \left\lfloor A \cdot k \right\rfloor \right) \right\rfloor$$

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 $T[0\ldots m-1]$. 
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 $(Ak)$.

\[
\begin{array}{c}
1 0 1 1 0 0 1 \\
\times \\
1 1 0 1 0 1 1 \\
\hline
1 0 0 1 0 1 0 0 1 1 0 0 1 1
\end{array}
\]

$h(k)$
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 $h(k_1)$, $h(k_2)$, $h(3)$... fall in different and pairwise remote locations in the table.

$$
\begin{array}{c}
1 & 0 & 1 & 1 & 0 & 0 & 1 \\
\times & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1
\end{array}
$$

$h(k)$
Lets 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 $h(k_1)$, $h(k_2)$, $h(3)$… fall in different and pairwise remote locations in the table.

$\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 \\
\times & 1 & 1 & 0 & 1 & 0 & 1 & 1 \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
\end{array}$

$h(k)$
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 \(h(k_1), \ h(k_2), \ h(3)\ldots\) fall in different and pairwise remote locations in the table.

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

\[
\times \quad 1 \ 0 \ 1 \ 1 \ 0 \ 0 \ 1 \\
\hline
1 \ 0 \ 0 \ 1 \ 0 \ 1 \ 0 \ 0 \ 1 \ 1
\]

\[
A = k^h(k)
\]

**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 $h(k_1), h(k_2), h(3) \ldots$ fall in different and pairwise remote locations in the table.

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & = A \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & = k \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & \end{array}
\]

\[h(k)\]

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 \( h(k_1), h(k_2), h(3) \ldots \) fall in different and pairwise remote locations in the table.

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 1 & 0 & 0 & 1 & = A \\
1 & 1 & 0 & 1 & 0 & 1 & 1 & = k \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
\hline
\end{array}
\]

\( h(k) \)

**Modular wheel**
Multiple hash functions:

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

Let's think about a system with 8 disks. 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 maintain 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 $3n/8$ records.
  - Efficiency: Search time should be small. When searching for a key, should not have to check each individual disk

- Need to support: $\text{Insert}(k)/\text{delete}(k)/\text{find}(k)$.
- We don’t know the data in advance - changing dynamically.
- So once a new record appears, we need to decide which 3 disks will store it. Once a query find($k$) appear, need to be able to find these 3 disks.
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 \( A_1 \ldots A_3 \) for example \((3k) \mod 8, (5k) \mod 8, (7k) \mod 8\)

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 \( h_1(k) \). Insert \( k \) into disk whose index is \( h_1(k) \) (example \( k = 15, h_1(k) = (3k) \mod 8, \) so we store this record in disk \((45 \mod 8) = \text{disk } 5\)).
  Similarly insert copies of \( k \) into disks \( h_2(k), h_3(k) \).
- **Search**\((k)\): Compute \( h_1(k), h_2(k), h_3(k) \) and check these disks. If don’t find, it is either because was never inserted or due to disk failures.
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 
\[ h(k) = (Ak) \text{mod} \ m \]

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 \( a_1, a_2, a_3, a_4 \ldots \)

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

\[ h(\text{BDCZ}) = (a_1 \cdot 66 + a_2 \cdot 68 + a_3 \cdot 67 + a_4 \cdot 90) \ \text{mod} \ m \]

(m is the size of the hash table. The ascii value of ‘B’ is 66 and of ‘Z’ is 90).

- Computing all constants \( a_i \) is very simple. Pick random integers between \( l \) and \( m-1 \).
- Excellent in practice, and in theory
- Involve one pass of the file, in the case of very long keys.
Before any data item arrives, decide about the size $m$ of the hash table. $m$ should be prime, and $>2n$. Let $m \approx 2^{20} = 1M$

Pick at random constants $\vec{a} = (a_0, a_2, \ldots a_r)$. Each $a_i$ is picked individually at random uniformly $1 < a_i < m - 1$.

Now the first key $k$ arrive. Let's break it into pairs of characters.

$k$ = “According to section 1223(b) a nonprofit organization…”

Break into pairs of characters

$k = \text{Ac|co|rd|in|g |to| s|ct}$

\[
\begin{array}{cccccccc}
  k_0 & k_1 & k_2 & k_3 & k_4 & k_5 & k_6 & k_7 & k_8 \\
  \text{Ac} & \text{co} & \text{rd} & \text{in} & \text{g} & \text{to} & \text{s} & \text{ec} & \text{ti} \\
\end{array}
\]

where $k_0 = 'A' \cdot 256 + 'c' = 65 \cdot 256 + 99$.

Finally $h_{\vec{a}}(k) = \left( \sum_{i=0}^{r} a_i \cdot k_i \right) \mod m$
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)
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,…m-1
  Output: Always a legit index in the table T[ ]. a number in the range 0,1…m-1

E.g.
- \[ h(k,i) = (k+i) \mod m \]
- \[ h(k,i) = (k+i \cdot h_2(k)) \mod m \]
  here \( h_2(k) \) is some other hash function
- \[ f(k,i) = (k+i^2) \mod m \]

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,…
  otherwise etc for \( h(k,2) , h(k,3) , \ldots , h(k,m–1) \).

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
  otherwise etc for \( h(k,2) , h(k,3) , \ldots , h(k,m–1) \).
Example of Insertion

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

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

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,…
- etc for \( h(k,2), h(k,3), \ldots, h(k,m-1) \).
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

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,…
etc for $h(k,2), h(k,3), \ldots, h(k,m-1)$. 
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

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,…
- etc for \( h(k,2), h(k,3), \ldots, h(k,m-1) \).
Example of Insertion

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

- 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

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,…

etc for \( h(k,2), h(k,3), \ldots, h(k,m-1) \).
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

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,…
etc for \( h(k,2), h(k,3), \ldots, h(k,m-1) \).
Example of Insertion

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

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

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

- \textbf{insert(15)}. \( h(15,0)=7 \)

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,…

\vspace{1cm}

\begin{tabular}{cccc}
\hline
\( T \) & 0 & 1 & 2 \\
\hline
 & & & \\
 & & & \\
 & & & \\
 & & & \\
12 & & & \\
 & & & \\
 & & & \\
 & & & \\
\hline
\end{tabular}
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). \( h(15,0)=7 \)

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,…
etc for \( h(k,2), h(k,3), \ldots, h(k,m-1) \).
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)
  $h(20,1)=5$
  The first attempt ($i=0$) checks $T[4]$, but it is occupied. So the second attempt ($i=1$) checks cell $h(k+i)=h(20+1)$.

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,…
- etc for $h(k,2)$, $h(k,3)$, …, $h(k,m-1)$. 
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). \( h(15,0) = 7 \)
- insert(20). \( h(20,0) = 4 \) (collision)
  - \( h(20,1) = 5 \)

The first attempt (i=0) checks \( T[4] \), but it is occupied. So the second attempt (i=1) checks cell \( h(k+i) = h(20+1) \).

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, etc for \( h(k,2), h(k,3), \ldots, h(k,m-1) \).
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)
  \( h(20,1)=5 \)
  The first attempt (\( i=0 \)) checks \( T[4] \), but it is occupied. So the second attempt (\( i=1 \)) checks cell \( h(k+i)=h(20+1) \).

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

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,… etc for \( h(k,2), h(k,3), \ldots, h(k,m-1) \).
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)
  $h(20,1) = 5$
  The first attempt ($i=0$) checks $T[4]$, but it is occupied. So
  the second attempt ($i=1$) checks cell $h(k+i)=h(20+1)$.

• insert(23). $h(23,0) = 7$ (collision)
  $h(23,1) = 0$

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,…
- 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)
  - \( h(20,1)=5 \)
  The first attempt (\( i=0 \)) checks \( T[4] \), but it is occupied. So the second attempt (\( i=1 \)) checks cell \( h(k+i)=h(20+1) \).

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

- insert(28). \( h(28,0)=4 \) (collision):
  - \( h(28,1)=5 \) (collision);
  - \( h(28,2)=6 \)

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,… etc for \( h(k,2), h(k,3), \ldots, h(k,m-1) \).
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)
  
  \( h(20,1) = 5 \)

  The first attempt \((i=0)\) checks \( T[4] \), but it is occupied. So the second attempt \((i=1)\) checks cell \( h(k+i)=h(20+1) \).

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

- **insert(28)**. \( h(28,0) = 4 \) (collision):
  
  \( h(28,1) = 5 \) (collision);
  
  \( h(28,2) = 6 \)

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,… etc for \( h(k,2), h(k,3), \ldots, h(k,m-1) \).
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)
  - insert(12). \( h(12,0) = 4 \)
  - insert(15).
  - insert(20). \( h(20,0) = 4 \) (collision)
    \( h(20,1) = 5 \)
    The first attempt \( i=0 \) checks \( T[4] \), but it is occupied.
  - insert(23). \( h(23,0) = 7 \)
  - insert(28). \( h(28,0) = 4 \) (collision)

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.
Searching a key. Example on the same table

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

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

\[ \begin{align*}
T & \begin{array}{c}
23 & \text{0} \\
12 & \text{1} \\
20 & \text{2} \\
28 & \text{3} \\
15 & \text{4} \\
12 & \text{5} \\
28 & \text{6} \\
15 & \text{7}
\end{array}
\end{align*} \]

- insert(12). \( h(12,0) = 4 \)
- insert(15).
- insert(20). \( h(20,0) = 4 \) (collision)
  \( h(20,1) = 5 \)
  The first attempt (\( i = 0 \)) checks \( T[4] \), but it is occupied.
- insert(23). \( h(23,0) = 7 \)
- insert(28). \( h(28,0) = 4 \) (collision)

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.
Searching a key. Example on the same table

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

- insert(12). \( h(12,0) = 4 \)
- insert(15)
- insert(20). \( h(20,0) = 4 \) (collision)
  \( h(20,1) = 5 \)
  The first attempt \( i=0 \) checks \( T[4] \), but it is occupied.
- insert(23). \( h(23,0) = 7 \)
- insert(28). \( h(28,0) = 4 \) (collision)

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.

\[ T \]

<table>
<thead>
<tr>
<th></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Search uses the same probe sequence. We stop when either finding the key, or when hitting an empty cell.


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.
**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)

- insert(12). \( h(12,0) = 4 \)
- insert(15).
- insert(20). \( h(20,0) = 4 \) (collision) \( h(20,1) = 5 \)
  - The first attempt \( i=0 \) checks \( T[4] \), but it is occupied.
- insert(23). \( h(23,0) = 7 \)
- insert(28). \( h(28,0) = 4 \) (collision)

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>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Example: Search 28. First check \( h(28,0) = 4 \) but \( T[4] \neq 28 \). Next check \( h(28,1) = 5 \) but \( T[5] \neq 28 \). Note \( T[7] \neq 28 \) - success.

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

\begin{itemize}
  
  \item insert(12). \( h(12,0) = 4 \)
  
  \item insert(15).
  
  \item insert(20). \( h(20,0) = 4 \) (collision) \( h(20,1) = 5 \)
  The first attempt (\( i=0 \)) checks \( T[4] \), but it is occupied.
  
  \item insert(23). \( h(23,0) = 7 \)
  
  \item insert(28). \( h(28,0) = 4 \) (collision)
\end{itemize}

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.

\begin{equation}
\begin{array}{c|ccccccc}
  & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
  T & 23 & 12 & 20 & 28 & 15 & & & \\
\end{array}
\end{equation}
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>
<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>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Search uses the same probe sequence. We stop when either finding the key, or when hitting an empty cell.


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.
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)

Search uses the same probe sequence. We stop when either finding the key, or when hitting an empty cell.


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.
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)

Search uses the same probe sequence. We stop when either finding the key, or when hitting an empty cell.


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.

Next, delete 20.
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)

Search uses the same probe sequence. We stop when either finding the key, or when hitting an empty cell.

Next, delete 20.


Next, Search 28 again. The search wrongly stops at the empty cell that used to contain 28.

Error

Searching a key. Example on the same table

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

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

Next, delete 20.

Next, search 28 again. The search wrongly stops at the empty cell that used to contain 28.

Error:

Search(16). \( h(16, 0) = 0 \). \( T[0] \neq 16 \). Next check \( h(16, 1) = 5 \), but \( T[5] \neq 28 \). Search terminates - 16 not in table.

Solution: Place a dummy (NIL) to indicate that this cell was non-empty, and search should continue and treat this cell as ‘nonempty’ and continue the search. The search stops only when reaching a cell that is empty.
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>
<thead>
<tr>
<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>12</td>
<td>28</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next, delete 20.


Next, Search 28 again. The search wrongly stops at the empty cell that used to contain 28.

Error


Solution: Place a dummy (NIL) to indicate that this cell was non-empty, and search should continue and treat this cell as ‘nonempty’ and continue the search. The search stops only when reaching a cell that is empty.
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)

Search uses the same probe sequence. We stop when either finding the key, or when hitting an empty cell.

Next, delete 20.

Example, Search 28. First check \( h(28,0) = 4 \), but \( T[4] \neq 28 \). Next check \( h(28,1) = 5 \) but \( T[5] \neq 28 \).

Next, search 28 again. The search wrongly stops at the empty cell that used to contain 28.

Error

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.

Solution: Place a dummy (NIL) to indicate that this cell was non-empty, and search should continue and treat this cell as ‘nonempty’ and continue the search. The search stops only when reaching a cell that is empty.

When inserting a new key, we can replace the flag 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

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