basic **Hashing**

The 3 main operations on data structures:

\[
\text{SEARCH} \quad \text{INSERT} \quad \text{DELETE}
\]

\(O(1)\) expected with assumptions
Hashing

The 3 main operations on data structures:

SEARCH
INSERT
DELETE

Direct access table: good when keys are distinct &

come from a small distribution \( U \).

e.g. \( U = \{0, 1, 2, \ldots, m-1\} \)
Hashing

The 3 main operations on data structures:

- Search
- Insert
- Delete

Direct access table: good when keys are distinct & come from a small distribution $U$.

e.g. $U = \{0, 1, 2, \ldots, m-1\}$

Say $m = 74$. Use an array:

```
0 1 2  \ldots  \_\_  \_\_  \ldots  \_\_  73
0 1 2  \ldots  \_\_  \_\_  \ldots  \_\_  m-1
```
The 3 main operations on data structures: SEARCH, INSERT, DELETE

Direct access table: good when keys are distinct & come from a small distribution \( U \).

\[ U = \{0, 1, 2, \ldots, m-1\} \]

Say \( m = 74 \). Use an array: 

\[ T = \begin{array}{cccccc}
0 & 1 & 2 & \cdots & \phi & \phi \\
0 & 1 & 2 & \cdots & k & \cdots & \phi & 73 \\
\end{array} \]

search\((T, 2)\) = 2
The 3 main operations on data structures: search, insert, delete.

Direct access table: good when keys are distinct & come from a small distribution U.

E.g. $U = \{0, 1, 2, \ldots, m-1\}$

Say $m=74$. Use an array: $T[0,1,2,\ldots,K,\phi,\ldots,\phi,73]$

Search $(T, 2) = 2$ // Insert $(T, k) \rightarrow T[k] = k$
**Hashing**

The 3 main operations on data structures: 

- **Search**
- **Insert**
- **Delete**

Direct access table: good when keys are distinct & come from a small distribution $U$.

E.g. $U = \{0, 1, 2, \ldots, m-1\}$

Say $m = 74$. Use an array:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73
```

```
search(T, 2) = 2  \quad \text{// insert}(T, k) \rightarrow T[k] = k  \quad \text{// delete}(T, k) \rightarrow T[k] = \emptyset
```

All $\Theta(1)$
If $U$ is larger than our available storage, this is much worse.

$U = \text{integers w/ } \leq 5 \text{ digits}$
If $U$ is larger than our available storage, this is much worse.

$T: \{79123, 65112, \ldots, \phi, \phi, 12837, \ldots, 95617\}$

$0, 1, \ldots, m-1$

$U$: all possible keys

$S$: subset of $U$: the keys being used.

Integers w/ 5 digits
If $U$ is larger than our available storage, this is much worse.

$U$: all possible keys
$S$: subset of $U$: the keys being used.
If $U$ is larger than our available storage, this is much worse.

$T: 79123, 65112, \ldots, \phi, \phi, 12837, \ldots, 95617$

$0, 1, \ldots, m-1$

$U$: all possible keys
$S$: subset of $U$: the keys being used.

$h$: hash function
maps keys to $T$.

$h(12837) = 1$
$h(65112) = k$
I want to quickly access data for students. Student ID's are annoying.

I remember names & years.

\[ U \]

- \( S \): class list
- All possible name & year combinations

\[ T \]

0
1
2
\vdots
m-1
- take first letter of name, map to number: $L = \{1 \ldots 26\}$
- map year/level similarly: $Y = \text{sophomore} = 2$, \text{junior} = 3, etc.
- Take first letter of name, map to number $L = \{1 \ldots 26\}$
- Map year/level similarly: $Y = \text{sophomore} = 2$, junior = 3, etc.
- $h(\text{student}) = 10 \cdot L + Y$ → unique for any value in $\{1, Y\}$

\[ U \]

\[ S: \text{COMP 160} \]

\[ S: \text{class list} \]

\[ \text{All possible name} \]
\[ \text{& year combinations} \]

\[ T \]

\[ \begin{align*}
0 & \quad 1 & \quad 2 & \quad \vdots & \quad m-1 \\
\vdots & & & \vdots & \\
\end{align*} \]

\[ Y = \{0 \ldots 9\} \]
- Take first letter of name, map to number $L = \{1\ldots26\}$
- Map year/level similarly: $Y = \text{Sophomore} = 2$, junior = 3, etc.
- $h(\text{student}) = 10 \cdot L + Y \rightarrow$ unique for any value in $\{1, Y\}$

$U$: COMP 160 class list

Charlie, junior

All possible name & year combinations

$h = 30 + 3$

$T$

$Y = \{0\ldots9\}$

$m-1 < 270$
The function $h$ is defined as follows:

- Take the first letter of the name, map to a number from $L = \{1 \ldots 26\}$
- Map the year/level similarly: $Y = \text{sophomore} = 2$, $\text{junior} = 3$, etc.
- $h(\text{student}) = 10 \cdot L + Y$ → unique for any value in $\{1, Y\}$

$S$: COMP 160 class list

Charlie (junior)

$h = 30 + 3$

All possible name & year combinations

$T$

0

$Y = \{0 \ldots 9\}$

Problems?

casey 33

$m-1 < 270$
\( h = \begin{cases} 
\text{take first letter of name, map to number } L = \{1 \ldots 26\} \\
\text{map year/level similarly: } Y = \text{sophomore} = 2, \text{ junior} = 3, \text{ etc} \\
h(\text{student}) = 10L + Y \rightarrow \text{unique for any value in } \{L,Y\} 
\end{cases} \)

- Whole possible name & year combinations
- \( h = 30 + 3 \)
- \( Y = \{0 \ldots 9\} \)

Problems?
1. Some permanently empty slots
2. Other \( h = 33 \)?
I could be more careful and design a hash function that will use much more information (full name + year + ID + ???) so that every student maps to a unique slot.

But, that is a lot of work, involves looking at the input distribution very carefully, and besides I'd have to do it all over next year.

Instead the simple function can be used year after year, as long as I can confidently deal with collisions some way.
What if many keys map to same slot?

\[
\begin{align*}
h(65112) &= k \\
h(2315) &= k \\
h(89) &= k
\end{align*}
\]
What if many keys map to same slot? Make a linked list.

Time for operations?
What if many keys map to same slot?

Make a linked list.

Insert = $\Theta(1)$

Search/Delete* = $\Theta(n)$

* $O(1)$ if not accessing by key (i.e., via search)
What if many keys map to same slot?

Make a linked list.

- **Insert** = $\Theta(1)$
- **Search/Delete** = $\Theta(n)$

*Must be consistent for each key*
What if many keys map to same slot?

Make a linked list.

Insert = $\Theta(1)$
Search/Delete = $\Theta(n)$

Must be consistent for each key

$n > m$: COLLISIONS are unavoidable
What if many keys map to same slot?

Make a linked list.

Insert = $\Theta(1)$
Search/Delete = $\Theta(n)$

Must be consistent for each key

$n > m$: **COLLISIONS** are unavoidable

minimize collisions by spreading $S$ into $T$ evenly
What if many keys map to same slot?

- Make a linked list.
  - Insert = $\Theta(1)$
  - Search/Delete = $\Theta(n)$

Must be consistent for each key.

$|S| = n$

$n > m$: COLLISIONS are unavoidable

minimize collisions by spreading $S$ into $T$ evenly

- want random-looking $h()$ yet consistent/deterministic
What if many keys map to same slot?

Make a linked list.

Insert = $\Theta(i)$
Search/Delete = $\Theta(n)$

Must be consistent for each key.

For a random $h$, every slot is equally likely:

Probability two given keys collide = $\frac{1}{m}$
What if many keys map to same slot?

Make a linked list.

Insert = $\Theta(1)$
Search/Delete = $\Theta(n)$

Must be consistent for each key

For a random $h$, every slot is equally likely: simple uniform hashing

Probability two given keys collide = $\frac{1}{m}$

Average # keys per slot = $\frac{n}{m}$

average size of linked list.
What if many keys map to same slot?

Make a linked list.

Insert = $\Theta(1)$
Search/Delete = $\Theta(n)$

Must be consistent for each key

For a random $h$, every slot is equally likely:

- Simple uniform hashing
  - Probability two given keys collide = $\frac{1}{m}$
  - Average # keys per slot = $\frac{n}{m} = \alpha = \text{"load factor"}$
    - $\alpha$ is the average size of linked list.
$|S| = n$

\[
\frac{n}{m} = \alpha = \text{"load factor"}
\]

average size of linked list.
\(|S| = n\)

\(\frac{n}{m} = \alpha = \text{"load factor"}\)

average size of linked list.

Expected time of unsuccessful search = ?
\(|S| = n\)

\(\frac{n}{m} = \alpha = \text{"load factor"}\)

Average size of linked list.

**Expected time of unsuccessful search** = \(\Theta(1+\ldots)\)

`search(key) : h(key) \rightarrow \text{slot #}`

Assuming evaluating \(h()\) is \(O(1)\)
Expected time of unsuccessful search = $\Theta(1+\alpha)$

$\frac{n}{m} = \alpha = \text{"load factor"}$

average size of linked list.

Search: $h(key) \rightarrow$ slot # ; search list.
\(|S| = n\)

\(\frac{n}{m} = \alpha = \text{"load factor"}\)

average size of linked list.

\[ \text{Expected time of unsuccessful search} = \Theta(1+\alpha) \]

search \((\text{key})\) : \(h(\text{key}) \rightarrow \text{slot} \#j\) \[\text{search list.}\]

\[ \text{Expected time of successful search} = \Theta(1+\alpha) \]

expect to search \(\frac{1}{2}\) of a list
Choosing Hashing Functions depending on keys and m.
Choosing Hashing Functions depending on keys and m.

Objective: get uniform distribution of keys to slots - always
Choosing Hashing Functions depending on keys and $m$.

Objective: get uniform distribution of keys to slots - always

Ex: $h(k) = k \mod m$

"Division method"

How good is this?
Choosing Hashing Functions depending on keys and m.

Objective: get uniform distribution of keys to slots - always

Ex: \( h(k) = k \mod m \)  

"Division method"  

\[ \text{If } S = \text{integers then it's fine.} \]

...but if \( S = m \cdot i \text{ for } i = 1, 2, 3 \text{ etc} \) \underline{FAIL}
Choosing Hashing Functions depending on keys and m.

Objective: get uniform distribution of keys to slots - always

Ex: \( h(k) = k \mod m \) \( \{ \) If \( S = \) integers then it's fine.

“Division method” \( \) \( \} \) ... but if \( S = m \cdot i \) for \( i = 1, 2, 3 \) etc \[ \text{FAIL} \]

We don't want any specific input pattern to affect uniformity.

(similar to sorting algo not working well on pre-sorted data)
Choosing Hashing Functions depending on keys and m.

Objective: get uniform distribution of keys to slots - always

Ex: \( h(k) = k \mod m \)  
"Division method"  
...but if \( S = m \cdot i \) for \( i = 1, 2, 3 \) etc FAIL

We don't want any specific input pattern to affect uniformity.  
(similar to sorting algo not working well on pre-sorted data)

"Fails" if m has a small divisor.  e.g. for even m, if all keys are even, half of T: empty.
If \( m = 2^r \) then \( k \mod m = ? \)

\[ r = 6 : \quad k = 1011000111011010 \]
If $m = 2^r$ then $k \mod m = k \mod 2^r$ keeps only last $r$ digits

$r=6 : \quad k = 1011000111011010$
If \( m = 2^r \) then \( k \mod m = k \mod 2^r \) keeps only last \( r \) digits.

\( r = 6 : \quad k = 1011000111011010 \)

\( h \) depends on a small part of the input (key).
If $m = 2^r$ then $k \mod m = k \mod 2^r$ keeps only last $r$ digits

$r=6 : \quad k = 1011000111011010$

$h$ depends on a small part of the input (key)

So, at least choose $m$ : prime & not close to power of 2
Suppose $m = 2^r$, and we are using $w$-bit words (keys).

\[ h(k) = (A \cdot k) \mod 2^w \text{ right-shifted by } w-r \]

\rightarrow \text{some odd integer in } [2^{w-r}, \ldots, 2^w-1] \Rightarrow w\text{-bit # with leading } 1. 

\text{Heuristic: pick } A \text{ not close to any power of } 2.

\text{ex: } m = 2^3 : r = 3 \quad A = 1011001 \quad k = 1101011 \quad \Rightarrow A \cdot k = 1001010 \underline{0110011}

h(1101011) = 011

If we had $A = 2^{w-1} \rightarrow A \cdot k = 110101100000000$

or, $A = 2^5 \rightarrow A \cdot k = 001101011000000$

\text{Heuristic provides some "randomness" to the process.}
Resolving collisions w/ open addressing assuming $n \leq m$
Resolving collisions w/ open addressing assuming $n \leq m$

The point is to avoid auxiliary linked lists.

one reason:

use as much space as possible

in advance for the table itself
Resolving collisions w/ open addressing assuming $n \leq m$

The point is to avoid auxiliary linked lists.
Instead, create a probe sequence as a function of key value.

$\Rightarrow$ permutation of slots to try.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>103</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>2014</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Resolving collisions w/ open addressing assuming $n \leq m$

The point is to avoid auxiliary linked lists.

Instead, create a probe sequence as a function of key value.\[\Rightarrow \text{permutation of slots to try.}\]

works because table is large
Resolving collisions with open addressing assuming $n \leq m$

The point is to avoid auxiliary linked lists. Instead, create a probe sequence as a function of key value. A permutation of slots to try. Works because table is large

ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6$. 
RESOLVING COLLISIONS w/ OPEN ADDRESSING assuming $n \leq m$

The point is to avoid auxiliary linked lists.
Instead, create a probe sequence as a function of key value.
\[ \text{permutation of slots to try.} \]
works because table is large

ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6.$

Insert(64): Try $T[9]: \text{full}$
Resolving Collisions w/ Open Addressing assuming n ≤ m

The point is to avoid auxiliary linked lists. Instead, create a probe sequence as a function of key value.

permutation of slots to try. works because table is large

ex: \( h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6 \).

Insert(64): Try \( T[9] \): full
& Try \( T[2] \): full
Resolving Collisions w/ Open Addressing assuming \( n \leq m \)

The point is to avoid auxiliary linked lists.

Instead, create a probe sequence as a function of key value.

\[ \text{permuation of slots to try.} \]

works because table is large

\[ \text{ex: } h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6. \]

Insert(64):

\[ \text{Try } T[9]: \text{full} \]
\[ \text{Try } T[2]: \text{full} \]
\[ \text{Try } T[4]: \text{full} \]
Resolving collisions w/ open addressing assuming \( n \leq m \)

The point is to avoid auxiliary linked lists.
Instead, create a probe sequence as a function of key value.

\[ \text{permutation of slots to try} \]

This works because table is large

![ Probe sequence example ]

\[ \text{ex: } h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6. \]

Insert(64):
- Try \( T[9] \): full
- Try \( T[2] \): full
- Try \( T[4] \): full
- Try \( T[8] \): ok
Resolving collisions w/ open addressing assuming \( n \leq m \)

The point is to avoid auxiliary linked lists.

Instead, create a probe sequence as a function of key value.

\[ \text{permutation of slots to try.} \]

works because table is large

For example, \( h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 2, 7, 10, 5, 6. \)

Insert(64):

- \( T[9] \): full
- \( T[2] \): full
- \( T[4] \): full
- \( T[8] \): ok

Search(64) follows same sequence.

Would return "not found" after 4 attempts.
ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6.$

Really this is $h(k, i)$
ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6.$

Really this is $h(k, i)$, $h(64, 1) = 9$.
ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6.$

Really this is $h(k, i) \quad h(64, 1) = 9 \quad h(64, 2) = 2$
ex: $h(64) \rightarrow 9,2,4,8,1,3,11,7,10,5,6.$

Really this is $h(k, i)$

$h(64, 1) = 9 \quad h(64, 2) = 2 \quad h(64, 3) = 4 \quad \text{etc}$
ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6.$
Really this is $h(k, i)$: $h(64, 1) = 9$ / $h(64, 2) = 2$ / $h(64, 3) = 4$ / etc.

Notice I didn't mention Delete.

Delete(64) : ?
ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6$.

Really this is $h(k, i)$  $h(64, 1) = 9$  $h(64, 2) = 2$  $h(64, 3) = 4$  etc.

Notice I didn't mention Delete.

Delete(64): $h(64, 1) = 9$, occupied by 2014
ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6$.

Really this is $h(k, i)$

\[
h(64, 1) = 9 \quad / \quad h(64, 2) = 2 \quad / \quad h(64, 3) = 4 \quad / \quad \text{etc}
\]

Notice I didn’t mention Delete.

Delete(64): $h(64, 1) = 9$, occupied by 2014

$h(64, 2) = 2$, occupied by 43
ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6$. 
Really this is $h(k, i)$ where $h(64, 1) = 9$, $h(64, 2) = 2$, $h(64, 3) = 4$, etc.

Notice I didn’t mention Delete. 

Delete(64) : $h(64, 1) = 9$, occupied by 2014 
$h(64, 2) = 2$, occupied by 43 
$h(64, 3) = 4$, occupied by 78
example: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6.$

Really this is $h(k, i)$

$h(64, 1) = 9 / h(64, 2) = 2 / h(64, 3) = 4 / etc$

Notice I didn’t mention ‘Delete’.

Delete(64): $h(64, 1) = 9$, occupied by 2014
$h(64, 2) = 2$, occupied by 43
$h(64, 3) = 4$, occupied by 78
$h(64, 4) = 8$, found 64, DELETE IT.

OK?
ex: \( h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6. \)

Really this is \( h(k, i) \)  
\[ h(64, 1) = 9 \]
\[ h(64, 2) = 2 \]
\[ h(64, 3) = 4 \]
\[ \text{etc.} \]

Notice I didn’t mention Delete.

\( \text{Delete}(64): \ h(64, 1) = 9, \ \text{occupied by} \ 2014 \)
\[ h(64, 2) = 2, \ \text{occupied by} \ 43 \]
\[ h(64, 3) = 4, \ \text{occupied by} \ 78 \]
\[ h(64, 4) = 8, \ \text{found 64, DELETE IT.} \]

But what if \( h(103) \rightarrow 4, 8, 2, 7, 1, 10, 11, 3, 5, 9, 6 \)?
example: \( h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6 \).

Really this is \( h(k, i) \):

- \( h(64, 1) = 9 \)
- \( h(64, 2) = 2 \)
- \( h(64, 3) = 4 \)

Notice I didn't mention Delete.

\[
\text{Delete}(64) : \quad h(64, 1) = 9, \text{ occupied by 2014} \\
\text{ } \quad h(64, 2) = 2, \text{ occupied by 43} \\
\text{ } \quad h(64, 3) = 4, \text{ occupied by 78} \\
\text{ } \quad h(64, 4) = 8, \text{ found 64, DELETE IT}.
\]

But what if \( h(103) \rightarrow 4, 8, 2, 7, 1, 10, 11, 3, 5, 9, 6 \)?

\[
\text{Search}(103) : \quad h(103, 1) = 4, \text{ occupied by 78}
\]
ex: \( h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6. \)

Really this is \( h(k, i) \)

\[
\begin{align*}
  h(64, 1) &= 9 / h(64, 2) = 2 / h(64, 3) = 4 / \text{etc}
\end{align*}
\]

Notice I didn’t mention Delete.

\[
\text{Delete}(64) : h(64, 1) = 9, \text{ occupied by } 2014 \\
\quad h(64, 2) = 2, \text{ occupied by } 43 \\
\quad h(64, 3) = 4+, \text{ occupied by } 78 \\
\quad h(64, 4) = 8, \text{ found } 64, \text{ DELETE IT.}
\]

But what if \( h(103) \rightarrow 4, 8, 2, 7, 1, 10, 11, 3, 5, 9, 6 \) ?

\[
\text{Search}(103) : h(103, 1) = 4-, \text{ occupied by } 78 \\
\quad h(103, 2) = 8, \text{ empty : declare } 103 \text{ not in } T.
\]
ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6$.

Really this is $h(k, i)$  

$\quad h(64, 1) = 9 \quad h(64, 2) = 2 \quad h(64, 3) = 4 \quad \text{etc}$

Notice I didn’t mention Delete.

$\text{Delete}(64) : \quad h(64, 1) = 9$, occupied by 2014  
$\text{h}(64, 2) = 2$, occupied by 4-3  
$\text{h}(64, 3) = 4$, occupied by 78  
$\text{h}(64, 4) = 8$, found 64, DELETE IT.

But what if $h(103) \rightarrow 4, 8, 2, 7, 1, 10, 11, 3, 5, 9, 6$ ?  

$\text{Search}(103) : \quad h(103, 1) = 4$, occupied by 78  
$\quad h(103, 2) = 8$, empty: declare 103 not in T.

Could use special “deleted” markers, but search time increases
Typical probing sequences
Typical probing sequences

Linear probing: $h(k, i) = (h(k, 0) + i) \mod m$
Typical probing sequences

Linear probing: \( h(k, i) = (h(k, 0) + i) \mod m \) \( \sim h(k) \) and wrap around.
Typical probing sequences

Linear probing: $h(k,i) = (h(k,0) + i) \mod m \sim h(k)$ and wrap around.

...tends to generate clusters.

probability of extending a cluster is higher than landing elsewhere
Typical probing sequences

Linear probing: \( h(k,i) = (h(k,0) + i) \mod m \sim h(k) \) and wrap around.
... tends to generate clusters.

Double hashing: \( h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m \)
Typical probing sequences

Linear probing: \( h(k,i) = (h(k,0) + i) \mod m \) \( \sim h(k) \) and wrap around. ... tends to generate clusters.

Double hashing: \( h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m \)

"random" offset
Typical probing sequences

Linear probing: \( h(k, i) = (h(k, 0) + i) \mod m \sim h(k) \) and wrap around.
... tends to generate clusters.

Double hashing: \( h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m \)

Heuristic: choose \( m = 2^r \) & \( h_2(k) : \text{odd} \).
otherwise \( \frac{1}{2} \)-table empty
Typical probing sequences

Linear probing: \( h(k,i) = (h(k,0) + i) \mod m \) \( \sim h(k) \) and wrap around.
...tends to generate clusters.

Double hashing: \( h(k,i) = (h_1(k) + i \cdot h_2(k)) \mod m \)

Heuristic: choose \( m = 2^r \) & \( h_2(k) \) : odd.
otherwise \( \frac{1}{2} \) table empty

See also quadratic probing in CLRS
\( \Rightarrow \) still only generates \( m \) probe sequences

Double hashing generates \( O(m^2) \) probe sequences : better