basic Hashing

The 3 main operations on data structures:

\[
\text{SEARCH, INSERT, DELETE} \quad O(1) \quad \text{expected with assumptions}
\]
Hashing

The 3 main operations on data structures: SEARCH, INSERT, DELETE

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

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

\[
\begin{array}{ccccccccccc}
0 & 1 & 2 & \cdots & \phi & \phi & \cdots & \phi & 73 \\
0 & 1 & 2 & k & m-1
\end{array}
\]
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:

search($T, 2$) = 2
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
\hline
0 1 2    \ldots
0 1 2    \ldots
\hline
k \phi
```

Search($T, 2$) = 2  \quad \text{// insert($T, k$) $\rightarrow T[k] = k$}
Hashing

The 3 main operations on data structures:

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: $T[0, 1, 2, \ldots, \phi, \phi, \ldots, \phi, 73]$.

Search $(T, 2) = 2$ \hspace{1cm} Insert $(T, k) \rightarrow T[k] = k$ \hspace{1cm} Delete $(T, k) \rightarrow T[k] = \phi$

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

$U =$ integers w/ $\leq 5$ digits

just a regular array
If \( U \) is larger than our available storage, this is much worse.

\[
T = \begin{array}{cccccccc}
4123 & 65112 & \cdots & \phi & \phi & 12837 & \cdots & 95617 \\
0 & 1 & & & & & & \text{m-1}
\end{array}
\]

\( U \): all possible keys

\( S \): subset of \( U \): the keys being used.

integers with \( 5 \) 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.

$U$: 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.

\[ U \]

\[ S: \text{ Comp 160 class list} \]

All possible name & year combinations

\[ T \]

\[ 0 \]

\[ 1 \]

\[ 2 \]

\[ \vdots \]

\[ m-1 \]
• take first letter of name, map to number $L = \{1...26\}$
• map year/level similarly: $Y = \text{sophomore} = 2$, $\text{junior} = 3$, etc

$U$

$\mathcal{S} \subseteq \text{comp 160 class list}$

All possible name & year combinations

$T$

0

1

2

\vdots

\vdots

m-1
Let $h : \{1, \ldots, 26\} \times \{1, \ldots, m\} \rightarrow \mathbb{N}$ be defined as:

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

The set $S$ contains all possible name and year combinations.

The diagram shows a partitioning of the set $T$ into $m$ parts, with each part containing $n$ elements, where $n \cdot m < 270$. 

The circle labeled $U$ contains the class list $S$.
\[ h = \begin{cases} 
\text{take first letter of name, map to number } L = \{1 \ldots 26\} \\
\text{map year/level similarly: } Y = \text{sophomore} = 2, \text{ junior} = 3, \text{ etc} \\
h(\text{student}) = 10 \cdot L + Y \quad \text{unique for any value in } \{1, Y\}
\end{cases} \]

\[ h = 30 + 3 \]

\[ Y = \{0 \ldots 9\} \]

\[ m-1 < 270 \]
We resume with the example of mapping class records to an array, based on the first letter of a student's first name, and the year they are in.

There is a problem if several students map to the same array slot.

This is unavoidable if the array is smaller than the number of students. Even if it is bigger though, the function is so simple that there can be "collisions". 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, requires 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.
\[ h = \{ \begin{align*}
& \text{take first letter of name, map to number } \quad L = \{1, \ldots, 26\} \\
& \text{map year/level similarly: } \quad Y = \text{sophomore} = 2, \text{ junior} = 3, \text{ etc} \\
& \text{h(student)} = 10 \cdot L + Y \quad \rightarrow \text{unique for any value in } \{L, Y\}
\end{align*} \]

- S: COMP 160 class list
- cyrus, junior

All possible name & year combinations

PROBLEMS?

1. Some permanently empty slots
2. Other \( h = 33 \)?

\[ m - 1 < 270 \]
What if many keys map to same slot?

\[ h(65112) = k \]
\[ h(2315) = k \]
\[ h(89) = k \]

\[ \{ \text{T}(k) = ? \} \]
What if many keys map to same slot?

Make a linked list.

Insert = Θ(1)
Search/Delete* = Θ(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 = $\Theta(i)$
- 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.

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

Try to 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 = $\Theta(1)$
- 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 = $\frac{1}{m}$
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:

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

Average # keys per slot = $\frac{n}{m} = \alpha$ = "load factor"

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 = \(\Theta(1)\)

\[\text{search(key)} : h(key) \rightarrow \text{slot#} ;\]

assuming evaluating \(h()\) is \(O(1)\)
$|S| = n$

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

average size of linked list.

Expected time of unsuccessful search = $\Theta(1+\alpha)$

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

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

average size of linked list.

Expected time of \underline{unsuccessful search} = \(\Theta(1+\alpha)\)

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

\(\text{search list.}\)
\[ |S| = n \]

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

average size of linked list.

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

\[ \text{search}(\text{key}) : h(\text{key}) \rightarrow \text{slot} \# ; \]

search list.

Expected time of successful search = \( \Theta(1+\alpha) \)

\[ \text{great if } \alpha = \Theta(1) \]

expect to search \( \frac{1}{2} \) list

see CLRS (overcomplicated?)
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

\[ h(k) = k \mod m \]  \text{if } S = \text{integers then it's fine.}

"Division method"  \text{...but if } S = m \cdot i \text{ for } i = 1, 2, 3 \text{ etc. 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 **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: \( h(k) = k \mod m \) \( \) If \( S = \) integers then it's fine.

“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 $k \mod m = k \mod 2^r$ keeps only last $r$ digits.

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

\( r=6 \) : \( k = 10110001111011010 \)

\( h \) 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
"Multiplication method"

Suppose $m=2^r$, and we are using $w$-bit words.
"Multiplication Method"

Suppose $m = 2^r$, and we are using $w$-bit words.

ex: $m = 2^3 : r = 3$
$w = 7$
$k = 1101011$
"Multiplication Method"

Suppose \( m = 2^r \), and we are using \( w \)-bit words.

Some odd integer in \([2^w - 1, 2^w - 1]\) \(\rightarrow\) \( w \)-bit # with leading 1.

- \(1000000\)
- \(1111111\)

Ok, and ending in a 1 too if odd

ex: \( m = 2^3 : r = 3 \)

\( w = 7 \)

\( A = 1011001 \)

\( k = 1101011 \)
"Multiplication Method"

Suppose \( m = 2^r \), and we are using \( w \)-bit words.

some odd integer in \([2^{w-1} \cdots 2^w - 1]\) \(\rightarrow\) \( w \)-bit \# with leading 1.

heuristic: pick A not close to any power of 2

ex: \( m = 2^3 \cdot r = 3 \)

\( w = 7 \)

\( A = 1011001 \)

\( k = 1101011 \)
"Multiplication Method"

Suppose $m = 2^r$, and we are using $w$-bit words.

Some odd integer in $[2^{w-1}, \ldots, 2^w - 1] \Rightarrow w$-bit # with leading 1.

Heuristic: pick $A$ not close to any power of 2

ex: $m = 2^3 : r = 3$  
$w = 7$  
$A = 1011001$  
$k = 1101011$  
$A \cdot k = 10010100110011$
"Multiplication Method"

Suppose $m = 2^r$, and we are using $w$-bit words.

$$(A \cdot k) \mod 2^w$$

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

Heuristic: pick $A$ not close to any power of 2

ex: $m = 2^3 : r = 3$
$w = 7$

$A = 1011001$
$k = 1101011$

$A \cdot k = 10010100110011$

remains after mod $2^7$
"Multiplication Method"

Suppose \( m = 2^r \), and we are using \( w \)-bit words. \((A \cdot k) \mod 2^w\) right-shifted by \( w-r \)

\( \xrightarrow{\text{some odd integer in } [2^{w-1}, 2^w-1]} \) \( w \)-bit # with leading 1.

Heuristic: pick \( A \) not close to any power of 2.

\[ \begin{align*}
\text{ex: } m &= 2^3 : r = 3 \\
A &= 1011001 \\
k &= 1101011 \\
A \cdot k &= 1001010011001111 \\
\end{align*} \]
"Multiplication Method"

Suppose \( m = 2^r \), and we are using \( w \)-bit words.

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

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

Heuristic: pick \( A \) not close to any power of 2

ex: \( m = 2^3 \), \( r = 3 \)
\( w = 7 \)
\( A = 1011001 \) \( \Rightarrow A \cdot k = 1001010 \overbrace{0110011}^{w-r} \) remains after \( \mod 2^7 \)

\[ h(1101011) = 011 \]
"MULTIPLICATION METHOD"

Suppose $m = 2^r$, and we are using $w$-bit words.

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

Some odd integer in $[2^{w-r} \ldots 2^{w-1}] \mapsto$ $w$-bit # with leading 1.

Heuristic: pick $A$ not close to any power of 2

ex: $m = 2^3 \land r = 3$

$A = 1011001$

$k = 1101011$

$A \cdot k = 1001010 \underbrace{0110011}_{w-r}$

remains after $\mod 2^7$

$h(1101011) = 011$

If we had $A = 2^{w-1}$ \rightarrow $A \cdot k = 1101011\overbrace{0000000}$

or, $A = 2^5$ \rightarrow $A \cdot k = 0011010\overbrace{1100000}$

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.
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. 

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

\[ \text{ex: } h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6. \]
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. Let permutation of slots to try. Works because table is large.

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

Insert(64): Try $T[9]$ = 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. 

\( \rightarrow \) permutation of slots to try. works because table is large

Example:

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

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

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
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. 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
- 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

one reason: use as much space as possible in advance for the table itself

- 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

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

Really this is $h(k, i)$. 

<table>
<thead>
<tr>
<th></th>
<th>36</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>43</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>78</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>103</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>2014</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6$.

Really this is $h(k, i)$, $h(64, 1) = 9$. 

```
1 36
  43
2
3
  78
4
  5
  103
5  64
  2014
6
7
8
9
10
11
```
ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6.$

Really this is $h(k, i)$

$h(64, 1) = 9$  \hspace{1cm} $h(64, 2) = 2$
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.
**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`.

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

Really this is $h(k, i)$

$\begin{array}{l}
h(64, 1) = 9 \\
h(64, 2) = 2 \\
h(64, 3) = 4 \\
\end{array}$

Notice I didn't mention 'Delete'.

Delete(64): $h(64, 1) = 9$, occupied by 2014 so...
$h(64, 2) = 2$, occupied by 43 so...
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 so...

$h(64, 2) = 2$, occupied by 43 so...

$h(64, 3) = 4$, occupied by 78 so...
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 so...
\( h(64, 2) = 2 \), occupied by 43 so...
\( h(64, 3) = 4 \), occupied by 78 so...
\( h(64, 4) = 8 \), found 64, DELETE IT.

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

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

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

Notice I didn't mention Delete.

\( \) \( \) \( \) \( \) \( \)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>43</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>78</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>103</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2014</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( \text{Delete}(64) : \) \( h(64, 1) = 9 \), occupied by 2014 so...

\( h(64, 2) = 2 \), occupied by 43 so...

\( h(64, 3) = 4 \), occupied by 78 so...

\( h(64, 4) = 8 \), found 64, DELETE IT.

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

As mentioned in class, suppose you inserted 64, then inserted 103, then deleted 64, then searched for 103
ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6$.

Really this is $h(k, i)$: $h(64, 1) = 9 \quad h(64, 2) = 2 \quad h(64, 3) = 4 \quad \text{etc}$

Notice I didn’t mention Delete.

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

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

Search(103): $h(103, 1) = 4$, occupied by 78
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 so...

$h(64, 2) = 2$ , occupied by 43 so...

$h(64, 3) = 4$ , occupied by 78 so...

$h(64, 4) = 8$ , found 64, DELETE IT.

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

Search(103) : 

$h(103, 1) = 4$ , occupied by 78

$h(103, 2) = 8$ , empty : declare 103 not in T.
ex: $h(64) \rightarrow 9, 2, 4, 8, 1, 3, 11, 7, 10, 5, 6$.

Really this is $h(k, i)$

Notice I didn't mention Delete.

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

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

Search(103): $h(103,1) = 4$, occupied by 78
$h(103,2) = 8$, 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.
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 : \( h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m \)

"random" offset

Heuristic : choose \( m=2^r \) & \( h_2(k) \) : 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: \( h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m \)

"random" offset

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