The 3 main operations on data structures:

\[
\{ \text{SEARCH, INSERT, DELETE} \}
\]
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: 

\[
\begin{array}{cccccccccc}
\text{0} & \text{1} & \text{2} & \ldots & \emptyset & \emptyset & \ldots & \emptyset & \text{73} \\
\text{0} & \text{1} & \text{2} & \ldots & \text{k} & \text{k} & \ldots & \text{k} & \text{m-1} \\
\end{array}
\]

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

\[\text{All } \Theta(1)\]
If \( U \) is larger than our available storage, this is much worse.

\[
\begin{array}{ccccccc}
T & 79123 & 65112 & \phi & \phi & 12837 & \ldots & 95617 \\
0 & 1 & & & & m-1 \\
\end{array}
\]

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

Integers with \( S \) 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. 

$h$: 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.

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

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?

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 = $\Theta(n)$ (list size)

*$n > m$: COLLISIONS are unavoidable

- minimize collisions by spreading $S$ into $T$ evenly
- want random-looking $h()$ yet consistent/deterministic
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.

What if many keys map to same slot?

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

Must be consistent for each key.

Simple uniform hashing (even though your function is deterministic, the more random it appears to be, the more this analysis is applicable).
\[ |S| = n \]

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

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

average size of linked list.

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.

Objective: get uniform distribution of keys to slots - always

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

"Division method" \( \cdots \) but if \( S = m \cdot i \) for \( i = 1, 2, 3 \) etc \( \underline{\text{FAIL}} \)

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

\( \text{"Fails" if } m \text{ has a small divisor. E.g. for even } m, \text{ if all keys are even, half of } T \text{ empty.} \)
If \( m = 2^r \) then \( k \mod m = k \mod 2^r \) keeps only last \( r \) digits

\[ r=6 : \quad k = \overbrace{10110001111011010}^{h \text{ depends on a small part of the input (key)}} \]

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.

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

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

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

$\text{ex: } m = 2^3, r = 3$

$w = 7$

\[ A = \begin{array}{c} 1011001 \\ 1000000 \end{array} \]

\[ k = \begin{array}{c} 1101011 \end{array} \]

\[
A \cdot k = \begin{array}{c} 1001010 \end{array} \begin{array}{c} 0110011 \end{array}
\]

remains after

\[
\mod 2^7
\]

If we had $A = 2^{w-1}$ \[ A \cdot k = \begin{array}{c} 11010110000000 \end{array} \]

or, $A = 2^5$ \[ A \cdot k = \begin{array}{c} 01101011000000 \end{array} \]

Heuristic provides some "randomness" to the process.
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.

⇒ 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 $T[8]$ : 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 $h(k, i)$

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

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

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.

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

Really this is $h(k, i)$

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

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

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 \( h(k, i) \) \( h(64, 1) = 9 \) / \( h(64, 2) = 2 \) / \( h(64, 3) = 4 \) / 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 $h(k, i)$  
$h(64, 1) = 9$ / $h(64, 2) = 2$ / $h(64, 3) = 4$ / 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.

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

Linear probing: \( h(k,i) = (h(k_0) + i) \mod m \) \( \sim h(k) \) and wrap around.

\[ \text{...tends to generate clusters.} \]

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

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

"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