Suppose we want to store a list of student records.

Student object

<table>
<thead>
<tr>
<th>Data</th>
<th>Want searches to be quick. Which would make a good key?</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>name</td>
</tr>
<tr>
<td>name</td>
<td>year</td>
</tr>
<tr>
<td>year</td>
<td>courses</td>
</tr>
</tbody>
</table>

Suppose ID is 6 digits (ex: 123456)

Want to store records in a way to quickly search by ID:
ex: courses for student with ID=123456

- Array of size $10^6$. Why $10^6$? Highest ID = 999,999
  - Index = 123456
  - Problem? Mostly empty. Lots of wasted space!

  - Just use last 4 digits of ID
  - Ex: ID = 123456
  - Index = 3456

Problem? Two students may have the same last 4 digits.
Few issues:
1. (How to) choose key?
2. (How to) map keys to index?
3. (How to deal with) 2 keys that map to same index? (Collision)

- Keys:
  - Should be unique to each record (or close to unique)
  - Should be integers
  - If string, convert to int

- Map key to index; table size = n
  - Always want index to be in \([0, n-1]\)
  - How to ensure? mod

  \[
  \text{index} = \text{key} \mod n = \left\lceil \frac{\text{hash(key)}}{n} \right\rceil
  \]

  Many many variations of hash functions
  - We will mostly look at key mod n + some variations

3. Collisions:
   - What to do if 2 keys hash (map) to same index?

   1. Separate chaining - keep doubly linked list of all elements that hash to the same index
Suppose keys: 1, 4, 9, 14, 21, 4, 36, 49, 24

Time? \( m \) = \# of elements, \( n \) = table size

worst-case:

Find: \( O(m) \) (all elements hash to same index)

Insert: if duplicates OK (stored in separate nodes): \( O(1) \)

"want to store duplicates together: \( O(m) \)" requires \( \text{find}() \)

Remove: \( O(m) \) (requires \( \text{find}() \) )

average-case:

Find / Insert / Remove: \( O(1) \)

Idea: Elements typically spread out evenly over \( n \) indices.

\( m \) elements, \( n \) indices \( \Rightarrow O(1 + \frac{m}{n}) \)

But \( \frac{m}{n} \) will be bounded by a constant \( c \).

\( \text{ex.} \times 10000 \) elements, \( n = 1000 \)

\( c \approx 10 \)

\( \frac{m}{n} = \text{load} = \lambda \)

worst-case is very rare!!
Separate chaining requires Linked Lists, pointers, mod

Alternative: when collision occurs, try other indices

(2) Linear Probing (Open Addressing)

If \( \text{hash}(\text{key}) \) full, try:

\[
(\text{hash}(\text{key}) + 1) \mod n, (\text{hash}(\text{key}) + 2) \mod n, \ldots \]

\( \uparrow \) add offset \( \rightarrow \) always mod \( n \) to get value in \([0, n-1]\)

In general, try \( (\text{hash}(\text{key}) + i) \mod n \) for \( i = 0, 1, 2, 3, \ldots \)

\[ \text{Note: linear function} \]

\[ n = 10 \]

\[
\begin{array}{c|c}
0 & 19 \\
1 & 18 \\
2 & 29 \\
3 & 9 \\
4 & 38 \\
5 & \\
6 & 7 \\
7 & \\
8 & 49 \\
9 & 49 \\
\end{array}
\]

- \( \text{insert}(49) \Rightarrow 9 \) \( (\text{hash}(\text{key}) + 0) \mod n \)
  \( = (\text{key} \mod 10 + 0) \mod 10 \)
  \( = 9 \mod 10 \)

- \( \text{insert}(38) \Rightarrow 8 \)

- \( \text{insert}(19) \Rightarrow 9 \) \( \text{full so try:} \)
  \( (\text{hash}(\text{key}) + 1) \mod 10 \)
  \( = (9 + 1) \mod 10 = 0 \)

- \( \text{insert}(18) \Rightarrow 8 \) \( \text{full} \)
  \( (8 + 1) \mod 10 = 9 \) \( \text{(full)} \)
  \( (8 + 2) \mod 10 = 0 \) \( \text{(full)} \)
  \( (8 + 3) \mod 10 = 1 \)

- \( \text{insert}(29) \Rightarrow 7 \) \( \text{full} \)
  \( (7 + 1) \mod 10 = 8 \) \( \text{full} \)
  \( (7 + 2) \mod 10 = 9 \) \( \text{(full)} \)
  \( (7 + 3) \mod 10 = 0 \) \( \text{(full)} \)
  \( (7 + 4) \mod 10 = 1 \)
Problem 2. Makes clusters (long blocks of filled slots).
If \( \lambda = \frac{m}{n} > 0.5 \) can be shown to require 2.5 probes on average.

Primary clustering – long blocks of filled slots.

Instead. Quadratic probing:
If hash(key) full, try:
\[
(\text{hash(key)} + 1) \mod n, (\text{hash(key)} + 2) \mod n, \ldots
\]

Try \( h_i = (\text{hash(key)} + i^2) \mod n \) for \( i = 0, 1, 2, \ldots \)

\[\begin{array}{c|c}
0 & 19 \\
1 & 29 \\
2 & \\
3 & \\
4 & \\
5 & \\
6 & \\
7 & \\
8 & 49-\\
9 & \\
\end{array}\]

Not: \text{insert}(19) \Rightarrow 9 \text{ full}
try: \((19 \mod 10) + 1) \mod 10 = 10 \mod 10 = 0\]

\text{insert}(29) \Rightarrow 9 \text{ full } \text{ (same as 19)}
try: \((29 \mod 10) + 1) \mod 10 = 10 \mod 10 = 0 \text{ full}
\((29 \mod 10) + 4) \mod 10 = 13 \mod 10 = 3\]

\text{already in table}

Time?

Worst-case:

- Find: \( O(m) \) (not \( O(n) \)) (probe every element)
- Insert: duplicates OK (stored separately): \( O(m) \) (find empty slot)
  together: \( O(m) \) requires \( \text{find()} \)
- Remove? How to remove? Can we just remove the element? No.

ex: 19
    29
    39
    9

Suppose remove(49)

Now find(19): 19 mod 0 = 9, empty!
   return null

Instead for remove: mark as 'deleted'.
- Slot available for future insert but value is removed

9 49 'deleted = true'

Avg-Case: Find/Insert/Remove: O(1)
    Elements will be fairly spread out in table, so constant # of probes

Problem with both Linear + Quadratic Probing?

Secondary clustering - keys mapped to the same index
(ex. 49, 19, 29) follow same probe sequence (9, 0, 3, ... )