

CSE 332 Summer 2026

Lecture 9: Hashing 2

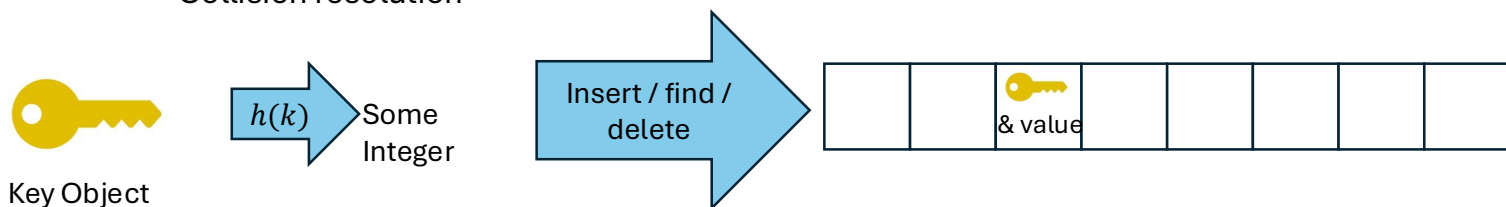
Michael Whitmeyer

<http://www.cs.uw.edu/332>

Hash Tables

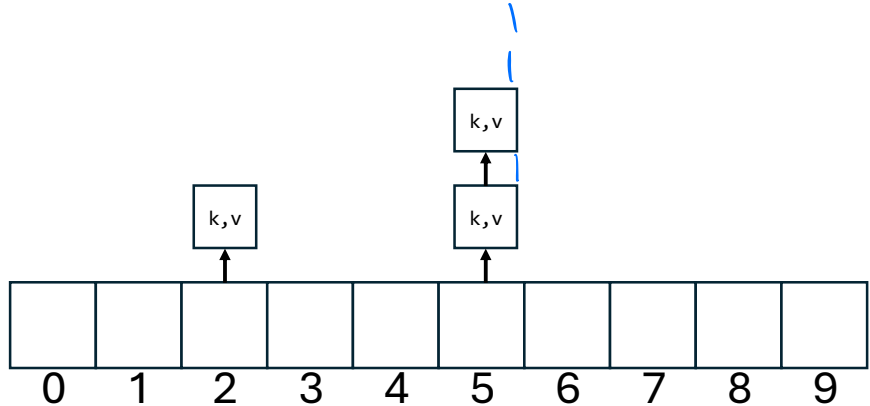
- Idea:

- Have a small array to store information
- Use a **hash function** to convert the key into an index
 - Hash function should “scatter” the keys, behave as if it randomly assigned keys to indices
- Store key at the index given by the hash function
- Do something if two keys map to the same place (should be very rare)
 - Collision resolution

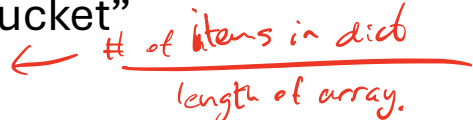


Separate Chaining Insert

- To insert k, v :
 - Compute the index using $i = h(k) \% \text{table.length}$
 - Add the key-value pair to the data structure at $\text{table}[i]$



Separate Chaining Runtime Analysis

- The **load factor** of a hash table represents the average number of items per “bucket”
 - $\lambda = \frac{n}{length}$  $\leftarrow \frac{\text{number of items in dict}}{\text{length of array.}}$
- Assume we have a hash table that uses a linked-list for separate chaining
 - What is the expected number of comparisons needed in an unsuccessful find?
 - What is the expected number of comparisons needed in a successful find?
- How can we make the expected running time $\Theta(1)$?

Separate Chaining Runtime Analysis (answers)

- The **load factor** of a hash table represents the average number of items per “bucket”

- $\lambda = \frac{n}{\text{length}}$

- Assume we have a hash table that uses a linked-list for separate chaining

- What is the expected number of comparisons needed in an unsuccessful find?

- λ

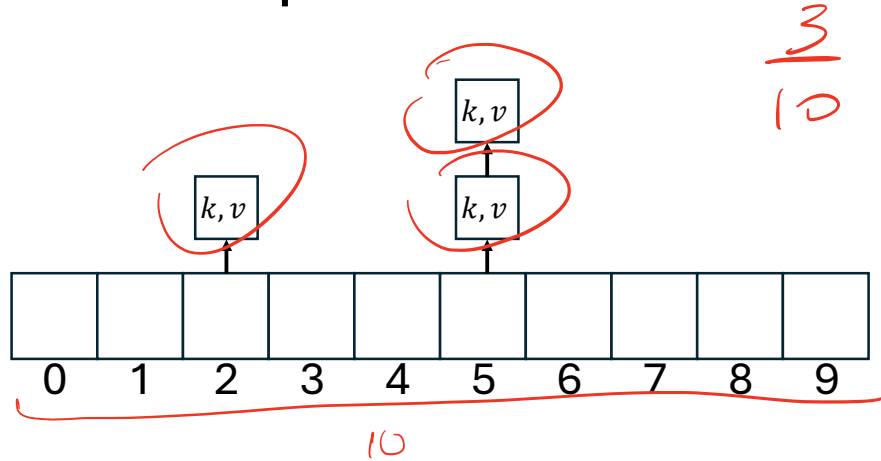
- What is the expected number of comparisons needed in a successful find?

- $\frac{\lambda}{2}$

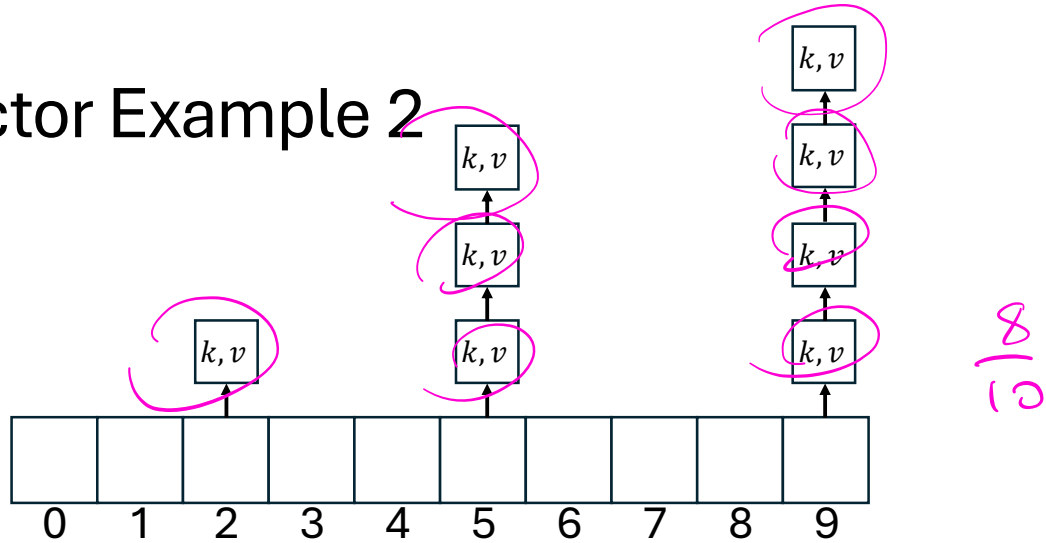
- How can we make the expected running time $\Theta(1)$?

- Pick a constant value, resize the array whenever λ exceeds that constant

Load Factor Example 1

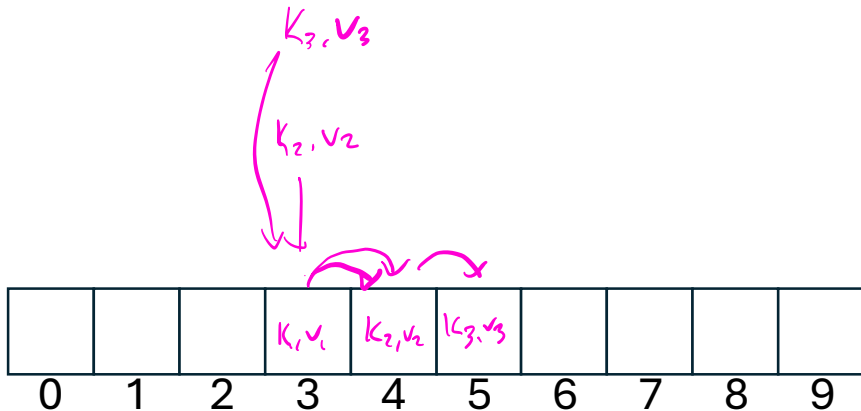


Load Factor Example 2



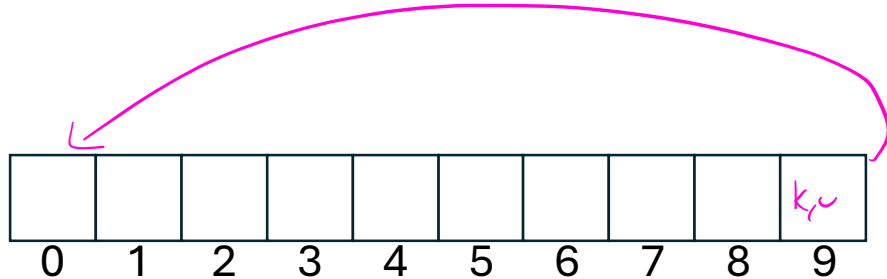
Collision Resolution: **Linear Probing**

- When there's a collision, use the next open space in the table



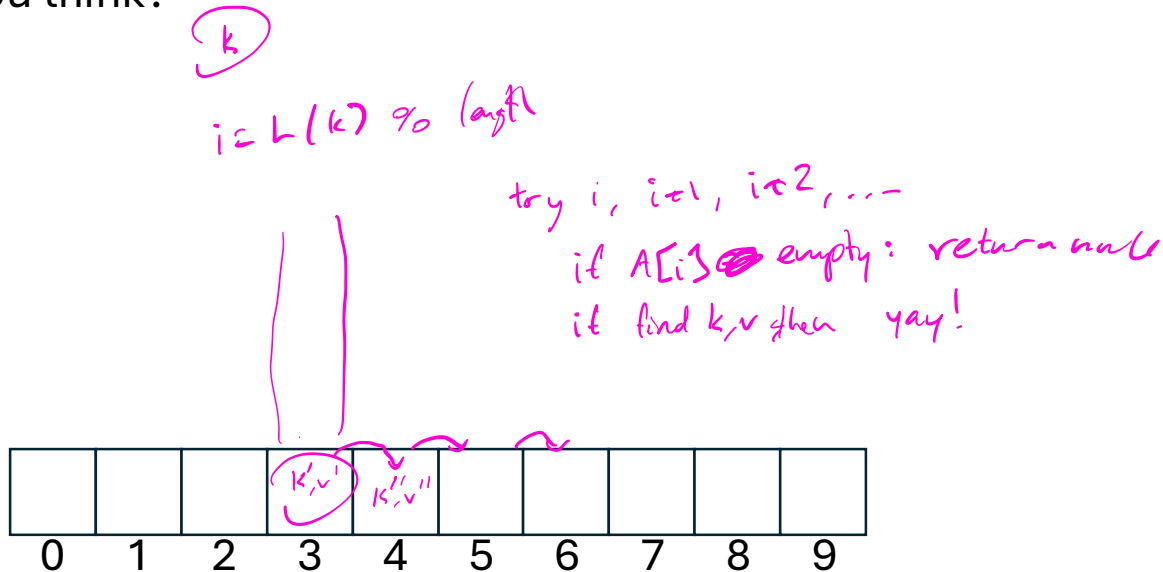
Linear Probing: Insert Procedure

- To insert k, v
 - Calculate $i = h(k) \% \text{table.length}$
 - If $\text{table}[i]$ is occupied then try index $(i+1) \% \text{table.length}$
 - If that is occupied try index $(i+2) \% \text{table.length}$
 - If that is occupied try index $(i+3) \% \text{table.length}$
 - ...



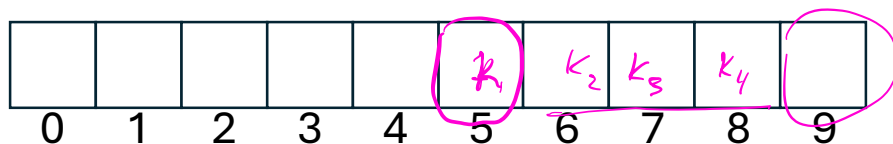
Linear Probing: How to find?

- What do you think?



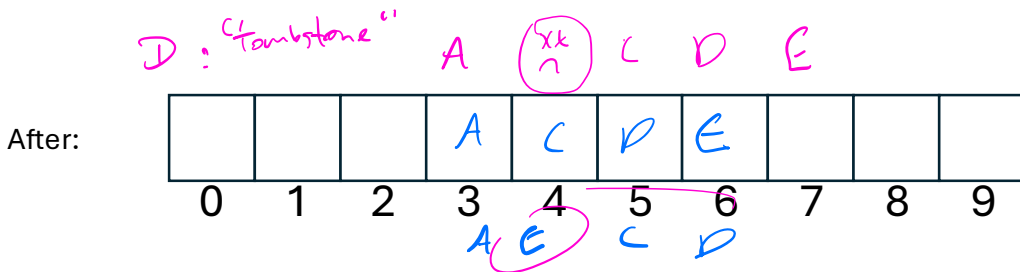
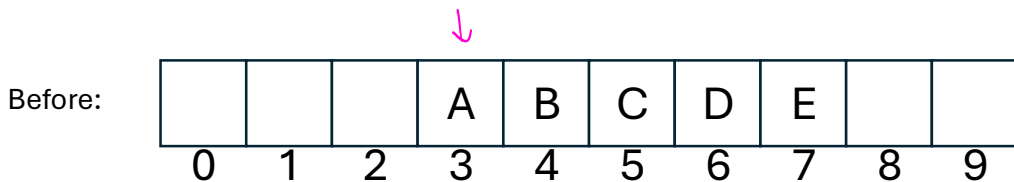
Linear Probing: Find

- To find key k
 - Calculate $i = h(k) \% \text{table.length}$
 - If $\text{table}[i]$ is occupied but doesn't have k , check $(i+1) \% \text{table.length}$
 - If that is occupied and doesn't contain k , check $(i+2) \% \text{table.length}$
 - If that is occupied and doesn't contain k , check $(i+3) \% \text{table.length}$
 - Repeat until you either find k or else you reach an empty cell in the table



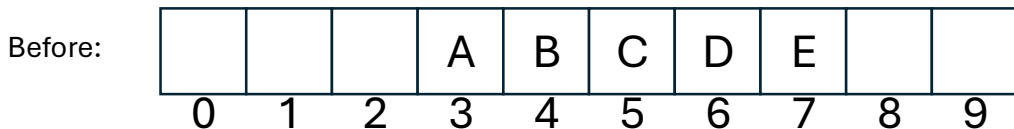
Linear Probing: Delete is Hard (Example 1)

- Suppose we insert A, B, C, D, and E in that order, where all map to 3
- How should we delete B?

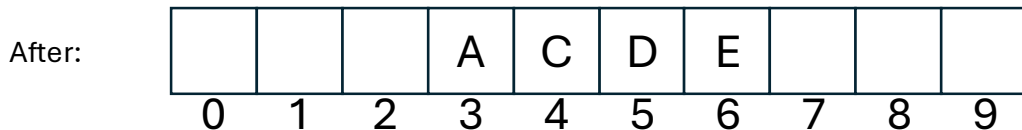


Linear Probing: Delete is Hard (Example 1)

- Suppose we insert A, B, C, D, and E in that order, where all map to 3
- How should we delete B?

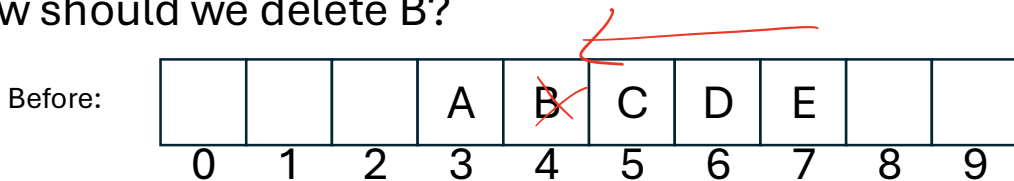


We need future probing to work correctly, so we must fill in the hole.

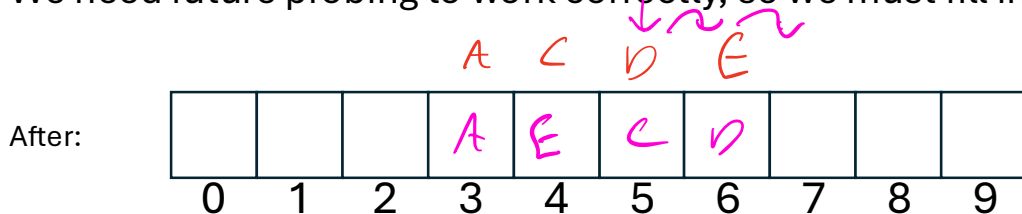


Linear Probing: Delete is Hard (Example 2)

- Suppose we insert A, B, C, D, and E in that order, where A,B,E all map to 3 and C,D map to 5.
- How should we delete B?



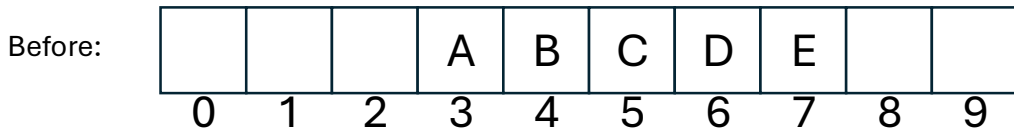
We need future probing to work correctly, so we must fill in the hole.



Linear Probing: Delete is Hard (Example 2)

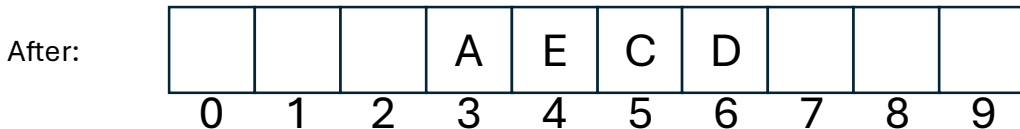
- Suppose we insert A, B, C, D, and E in that order, where A,B,E all map to 3 and C,D map to 5.
- How should we delete B?

C: 2, 6, 7, 8, 9, 0



We need future probing to work correctly, so we must fill in the hole.

We cannot move C or D over because they map to an index later in the probe path



$h(c) > 4$

Linear Probing: Delete is Hard (Example 3)

- Suppose we insert A, B, C, D, and E in that order, where A, B, E map to 8 and C, D map to 0.
- How should we delete B?

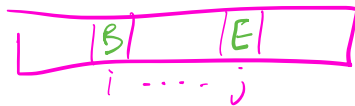
Before:

C	D	E						A	B
0	1	2	3	4	5	6	7	8	9

We cannot move C or D over because they map to an index later in the probe path, even though the index number happens to be smaller.

After:

C	D							A	E
0	1	2	3	4	5	6	7	8	9



probe path j : $i-2, i-1, i, \dots$

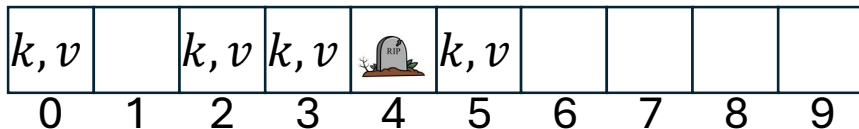
Linear Probing: Delete

- **Option 1 (harder):** Plug the hole with other items in a way that makes probes behave correctly

- Rough idea: to delete at index i , starting from i and going until we find an empty index, if probe path of the key at index j has index i before j then move the key-value pair at j to index i , then repeat on index j .

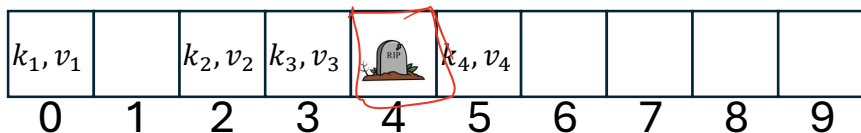
- **Option 2 (easier):** “Tombstone” deletion. Leave a special object that indicates an something was deleted from there

- The tombstone does not act as an open space when finding (so keep looking after its reached)
- When inserting you can replace a tombstone with a new item



Linear Probing + Tombstone: Find

- To find key k
 - Calculate $i = h(k) \% \text{table.length}$
 - While $\text{table}[i]$ has a key other than k , set $i = (i+1) \% \text{table.length}$
 - If you come across k return $\text{table}[i]$
 - If you come across an empty index, the find was unsuccessful
 - Tombstones do not count as empty!



Linear Probing + Tombstone: Insert

- To insert k, v
 - Calculate $i = h(k) \% \text{table.length}$
 - While $\text{table}[i]$ has a key other than k , set $i = (i+1) \% \text{table.length}$
 - If $\text{table}[i]$ has a tombstone, set $\text{dest} = i$
 - That is where we will insert if the find is unsuccessful
 - If you come across k , set $\text{table}[i] = k, v$
 - If you come across an empty index, the find was unsuccessful
 - Set $\text{table}[\text{dest}] = k, v$ if we saw a tombstone
 - Set $\text{table}[i] = k, v$ otherwise

k_1, v_1		k_2, v_2	k_3, v_3		k_4, v_4				
0	1	2	3	4	5	6	7	8	9

Downsides of Linear Probing



- What happens when λ approaches 1?

- Get longer and longer contiguous blocks
- A collision is guaranteed to grow a block
 - Larger blocks experience more collisions
 - Feedback loop!

$$\lambda = \frac{n}{\text{length}}$$

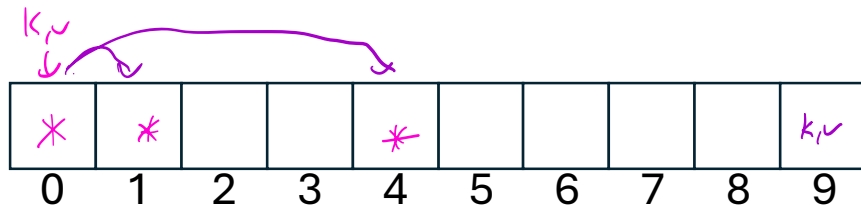
- What happens when λ exceeds 1?

- Impossible!
- You can't insert more stuff

Quadratic Probing: Insert Procedure

- To insert k, v
 - Calculate $i = h(k) \% \text{table.length}$
 - If $\text{table}[i]$ is occupied then try $(i+1^2) \% \text{table.length}$
 - If that is occupied try $(i+2^2) \% \text{table.length}$
 - If that is occupied try $(i+3^2) \% \text{table.length}$
 - If that is occupied try $(i+4^2) \% \text{table.length}$
 - ...

$$h(k) = 0 \% \text{length}$$



Quadratic Probing: Example

- Insert:

$$h(k) = k$$

- 76

- 40

- 48

- 5 (5)

- 55 (6)

- 47 (5)

FAIL

48		5	55		40	76
0	1	2	3	4	5	6

Using Quadratic Probing

- If you probe table.length times, you start repeating indices
- If table.length is prime and $\lambda < \frac{1}{2}$ then you're guaranteed to find an open spot in at most table.length/2 probes
proof on next slide
- ~~Helps with the clustering problem of linear probing, but does not help if many things hash to the same value~~



Quadratic Load Factor Lemma

Fact: if $a \cdot b \equiv 0 \pmod{p}$
then either $a \equiv 0 \pmod{p}$
or $b \equiv 0 \pmod{p}$

• **Lemma:** $1^2, 2^2, \dots, \left\lfloor \frac{p}{2} \right\rfloor^2$ are distinct mod p .

• **Proof:** $S = \{1, 2, \dots, \left\lfloor \frac{p}{2} \right\rfloor\}$

suppose towards a contradiction $\exists x, y \in S$ with $x \neq y$ s.t. $x^2 \equiv y^2 \pmod{p}$

$$x^2 - y^2 \equiv 0 \pmod{p}$$

$$(x-y)(x+y) \equiv 0 \pmod{p}$$

$\Rightarrow$ either ① $x-y \equiv 0 \pmod{p}$

or ② $x+y \equiv 0 \pmod{p}$

contradiction

$\Rightarrow 1^2, 2^2, 3^2, \dots, \left\lfloor \frac{p}{2} \right\rfloor^2$
are all distinct

$x-y \in \left\{ -\left\lfloor \frac{p}{2} \right\rfloor + 1, \dots, -1, 1, \dots, \left\lfloor \frac{p}{2} \right\rfloor \right\}$
contradiction.

Double Hashing: Insert Procedure

- Given h and g are both good hash functions
- To insert k, v
 - Calculate $i = h(k) \% \text{table.length}$
 - If $\text{table}[i]$ is occupied then try $(i + g(k)) \% \text{table.length}$
 - If that is occupied try $(i + 2 * g(k)) \% \text{table.length}$
 - If that is occupied try $(i + 3 * g(k)) \% \text{table.length}$
 - If that is occupied try $(i + 4 * g(k)) \% \text{table.length}$
 - ...

$$g(k) \neq 0$$

0	1	2	3	4	5	6	7	8	9

Rehashing

- If your load factor λ gets too large, copy everything over to a larger hash table
 - To do this: make a new, larger array
 - Re-insert all items into the new hash table by reapplying the hash function
 - We need to reapply the hash function because items should map to a different index
 - New array should be “roughly” double the length (but probably still want it to be prime)
- What does “too large” mean?
 - For separate chaining, typically we want $\lambda < 2$
 - For open addressing, typically we want $\lambda < \frac{1}{2}$

Sorting

- Rearrangement of items into some defined sequence
 - Usually: reordering a list from smallest to largest according to some metric
- Why sort things?
 - Enable things like binary search
 - It makes some algorithms faster
 - Nicer for human algorithms too
 - Data organization

More Formal Definition

- Input:
 - An array A of items
 - A comparison function for these items
 - Given two items x and y , we can determine whether $x < y$, $x > y$, or $x = y$
- Output:
 - A permutation of A , call it B , such that if $i \leq j$ then $B[i] \leq B[j]$
 - Permutation: a sequence of the same items but perhaps in a different order

Sorting “Landscape”

- There is no singular best algorithm for sorting
- Some are faster, some are slower
- Some use more memory, some use less
- Some are super extra fast if your data matches particular assumptions
- Some have other special properties that make them valuable
- No sorting algorithm can have only all the “best” attributes