Hashing

Part Two

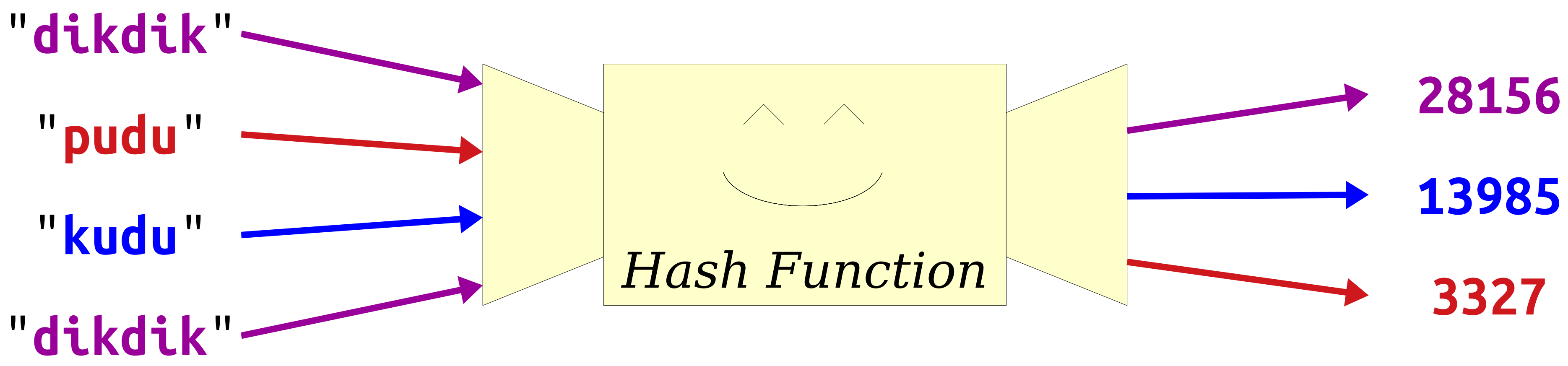
Outline for Today

- ***Recap from Last Time***
  - A quick refresher on hash functions.
- ***Hashing Variants***
  - We built a hash table last lecture. There are other strategies we could have used.
- ***Linear Probing***
  - A deceptively simple and fast hashing scheme.
- ***Robin Hood Hashing***
  - Moving items around in a hash table.

Recap from Last Time

Hash Functions

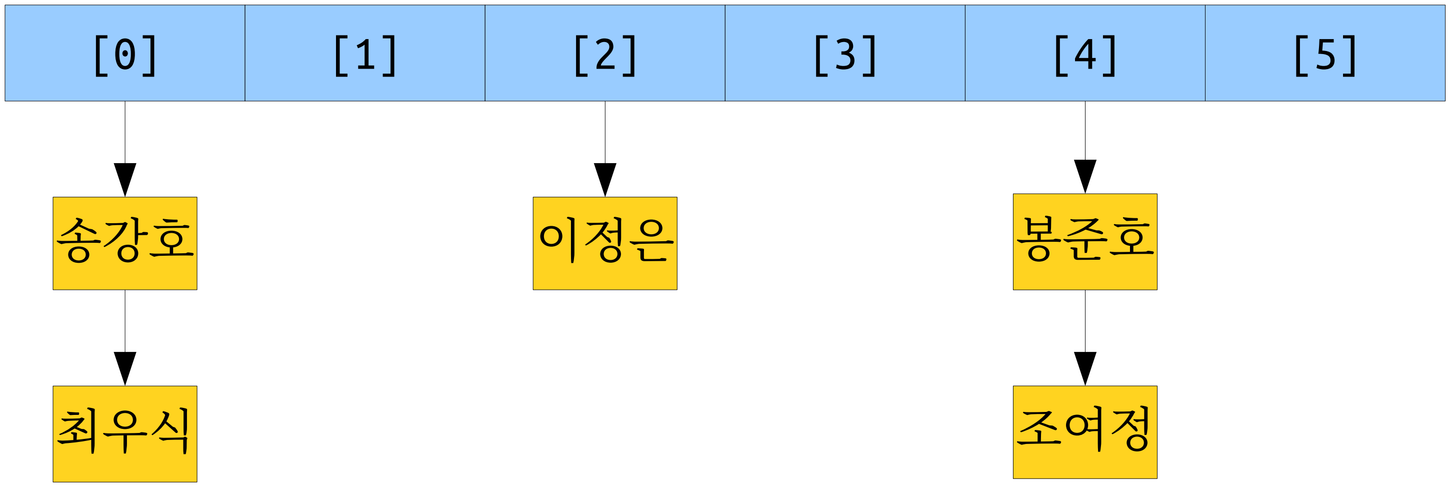
- A ***hash function*** is a function that takes an object as input and produces an integer called its ***hash code***.



- If you feed the same input to a hash function multiple times, it will always produce the same output.
- Aside from this, though, the outputs of hash functions should look more or less random.

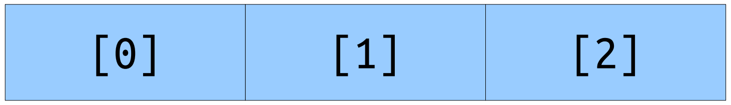
Hash Tables

- A ***hash table*** is a data structure where items are positioned in an array based on their hash code.
- Last time, we saw ***chained hashing***, where all items with the same hash code are stored in the same slot.



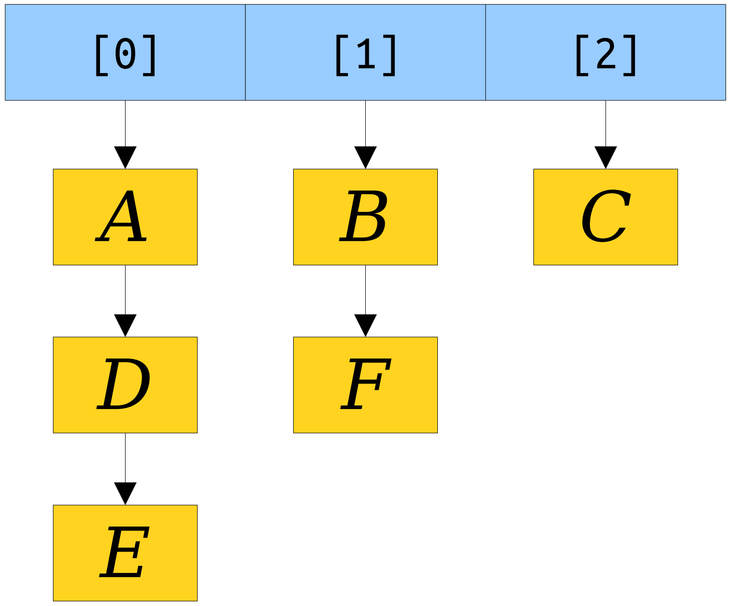
Try It Yourself!

- Insert the following values into this hash table.
  - *A* (hash code 0)
  - *B* (hash code 1)
  - *C* (hash code 2)
  - *D* (hash code 0)
  - *E* (hash code 0)
  - *F* (hash code 1)



Try It Yourself!

- Insert the following values into this hash table.
  - *A* (hash code 0)
  - *B* (hash code 1)
  - *C* (hash code 2)
  - *D* (hash code 0)
  - *E* (hash code 0)
  - *F* (hash code 1)



New Stuff!

Making Fast Hash Tables

- Hash tables, like the one we saw last time, are among the most-commonly-used data structures in practice.
- As a result, it's important for them to work as quickly and efficiently as possible.
- Anecdote: Google recently invested years of effort building a faster hash table. Why?

(*Better hash tables*) × **(*Lots of computers*)**

=

(*Huge equipment, power, and CO_2 savings*)

- The faster table they developed is based on insights from a different approach to building `Map` and `Set`.

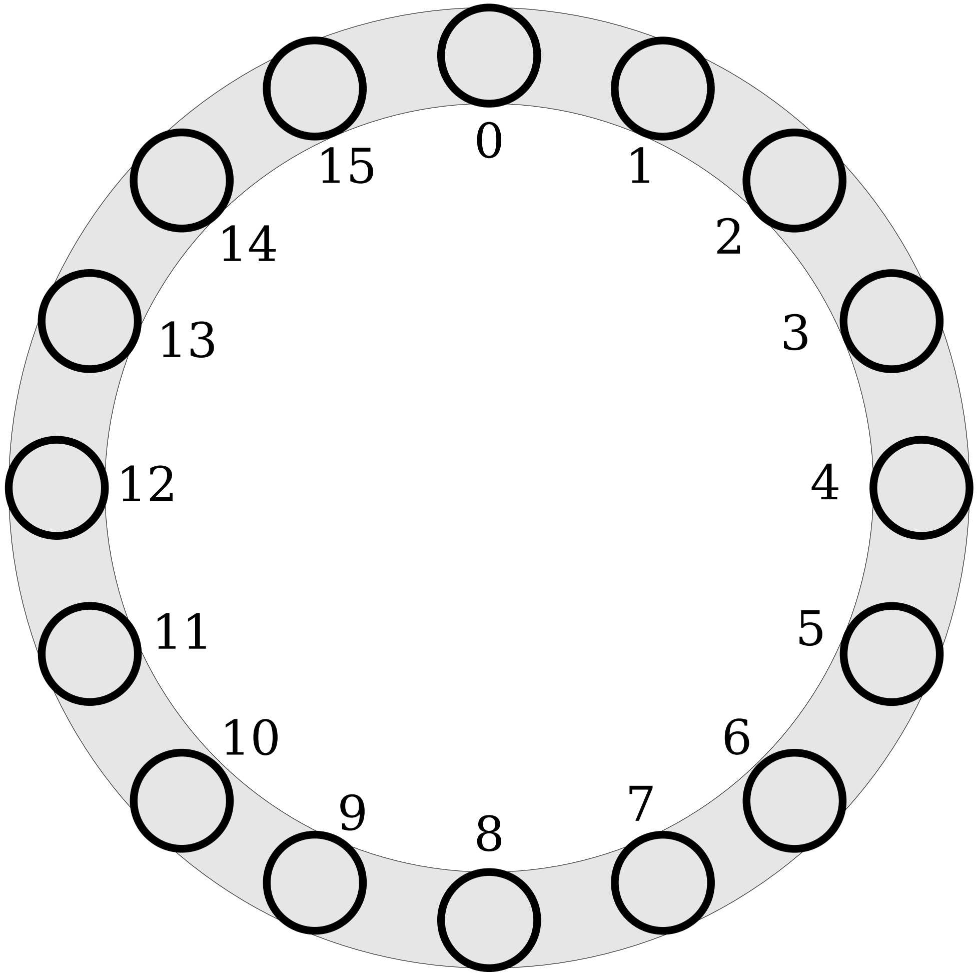
Open Addressing

- The style of hashing we saw last time is called ***chained hashing***, since we “chain” together all the items that have the same hash code.
- There is a family of other hash tables that use an idea called ***open addressing***.
- In open addressing,

  ☞ ***each table slot holds at most one element***. ☜
- If multiple elements hash to the same slot, they “leak out” and spill over into other free slots.
- These strategies form the basis for some of the fastest hash tables.

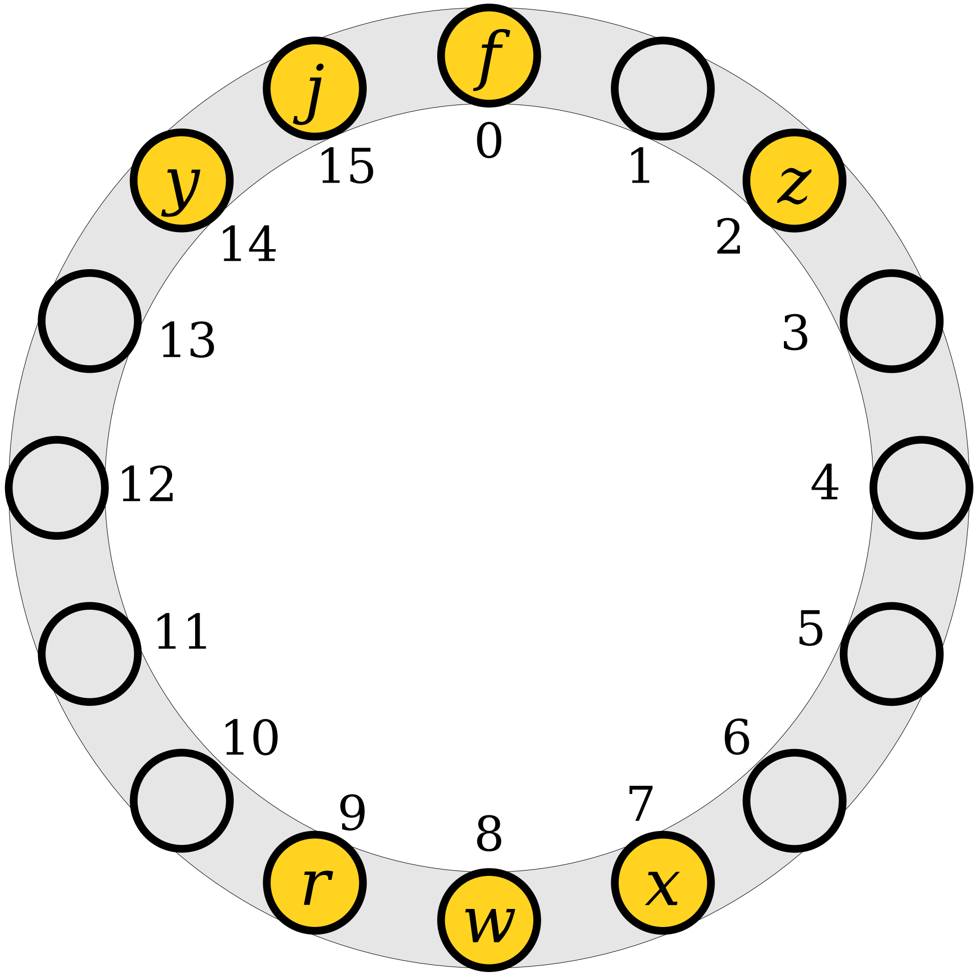
Linear Probing

- ***Linear probing*** is a simple open-addressing hashing strategy.
- We maintain an array of ***slots***, which we think of as forming a ring.



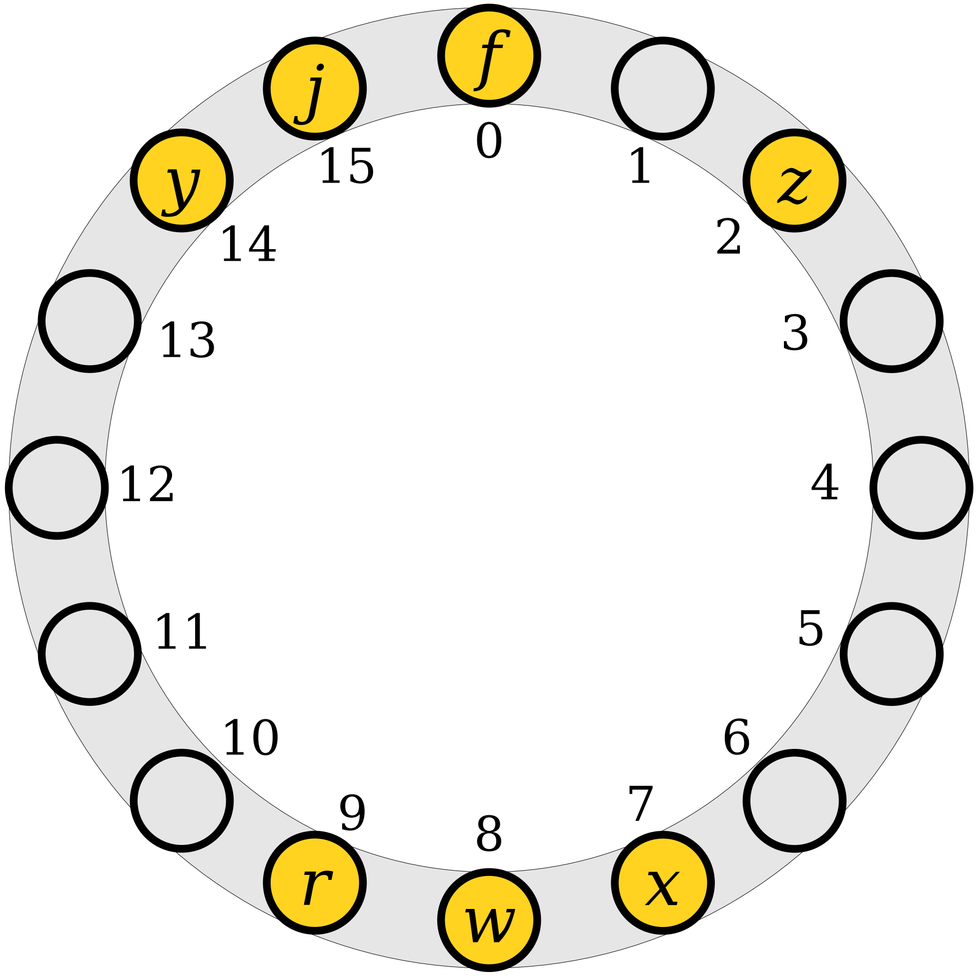
Linear Probing

- To insert an element, compute its hash code and try to place it at the slot with that number.
- If that spot is occupied, keep moving through the array, wrapping around at the end, until a free spot is found.



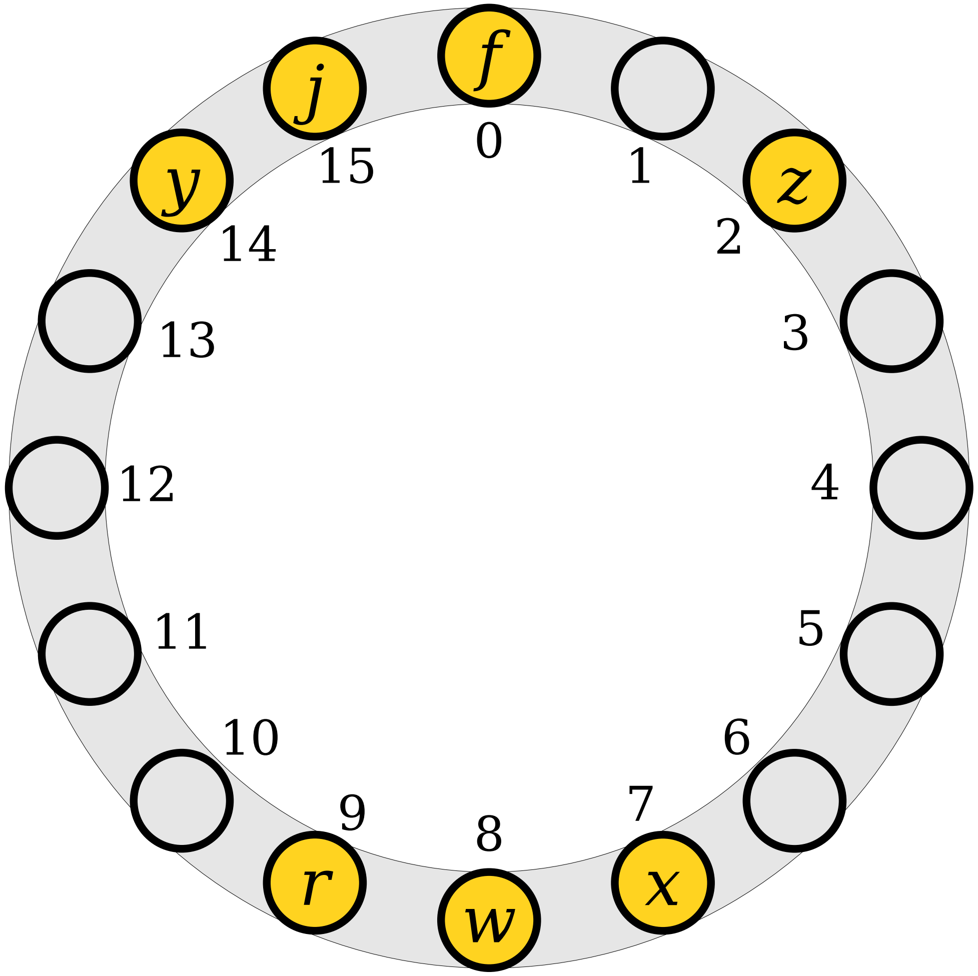
Linear Probing

- To look up an element, compute its hash code and start looking there.
- Move around the ring until either the element is found or a blank spot is detected.
- (If every single slot is full, stop looking after you've tried them all.)



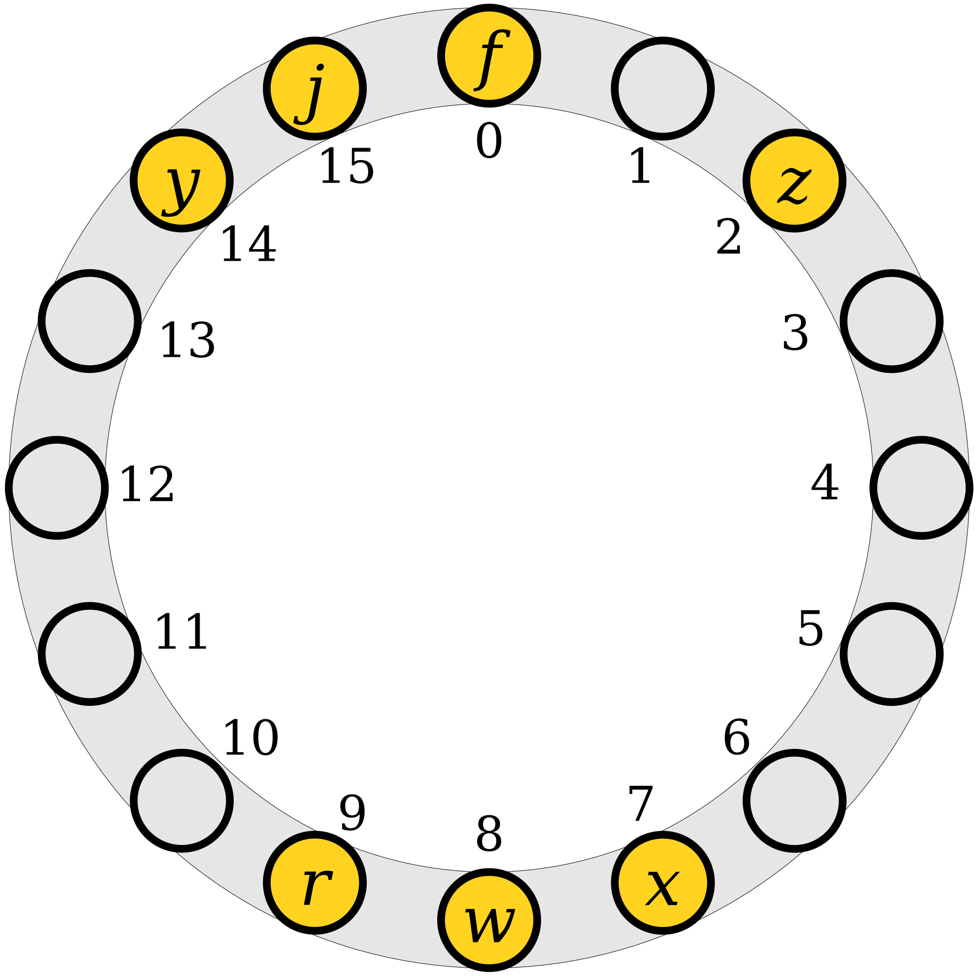
Linear Probing

- Deletions are a bit trickier than in chained hashing.
- We cannot just do a search and remove the element where we find it.
- Why?



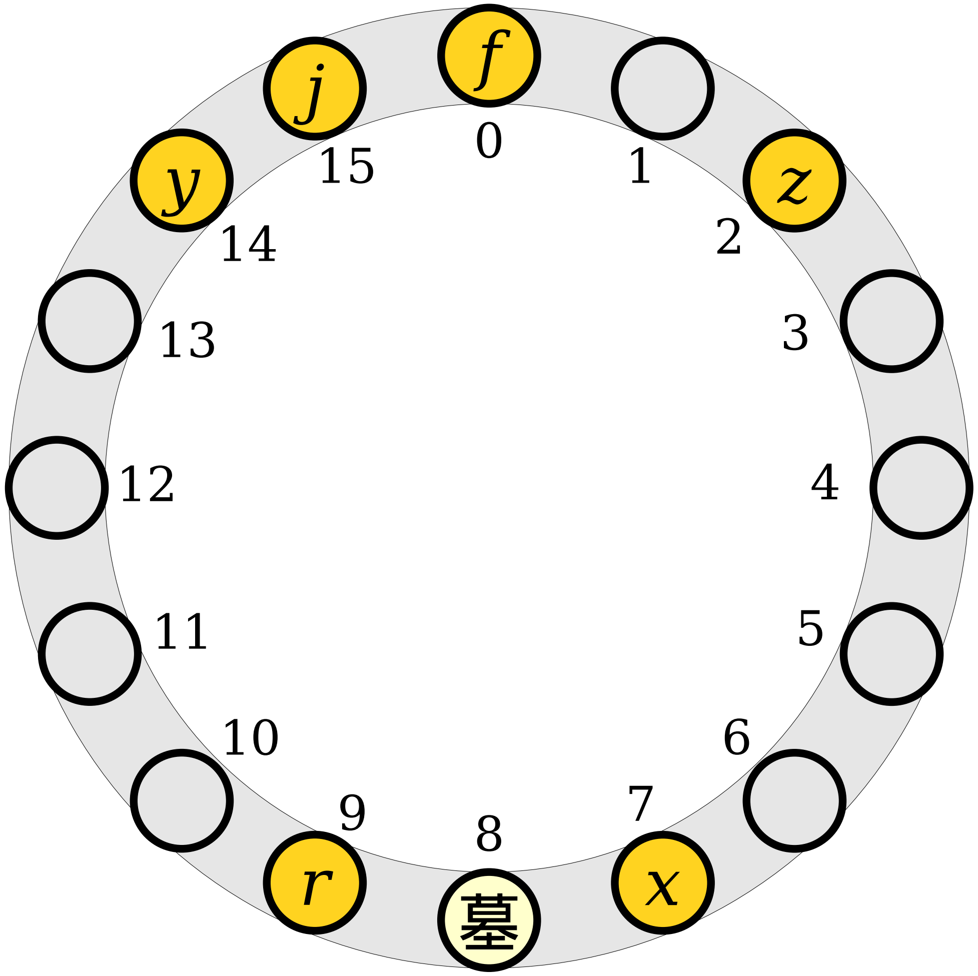
Linear Probing

- Deletions are often implemented using ***tombstones***.
- When removing an element, mark that the slot is empty and was previously occupied.
- When doing a lookup, don't stop at a tombstone. Instead, keep the search going.



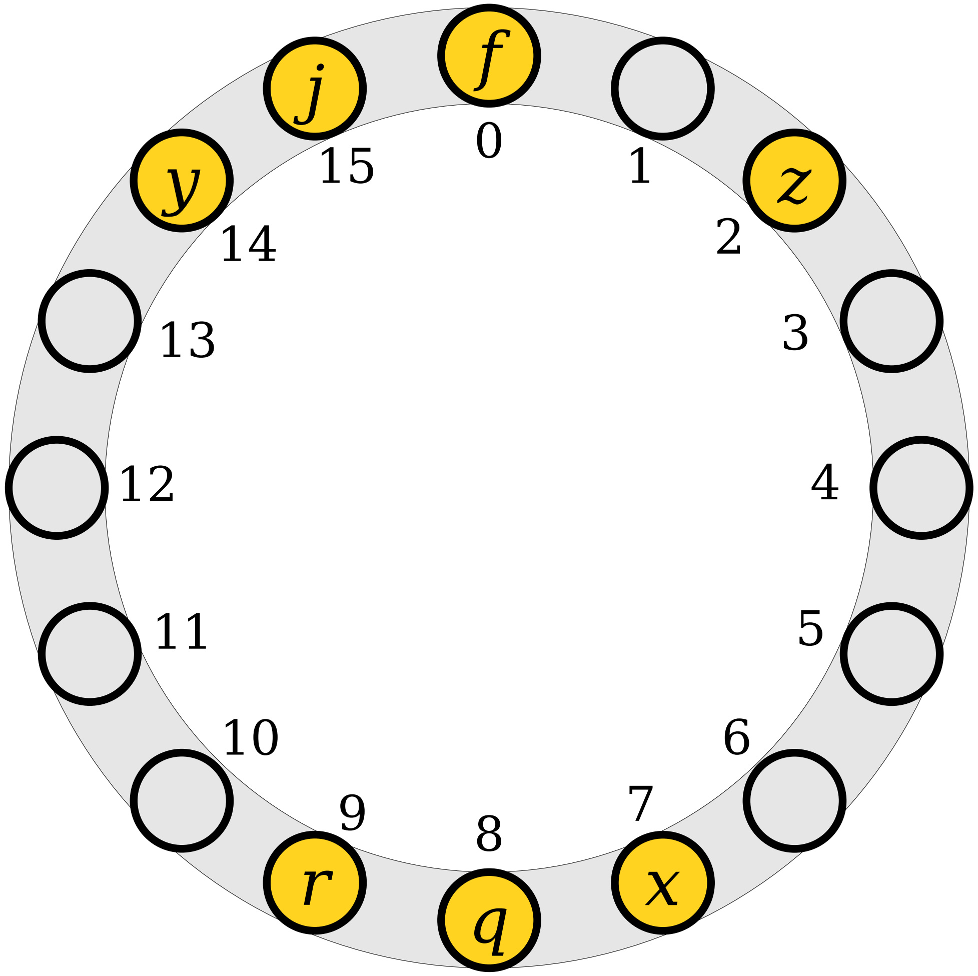
Linear Probing

- Deletions are often implemented using ***tombstones***.
- When removing an element, mark that the slot is empty and was previously occupied.
- When doing a lookup, don't stop at a tombstone. Instead, keep the search going.



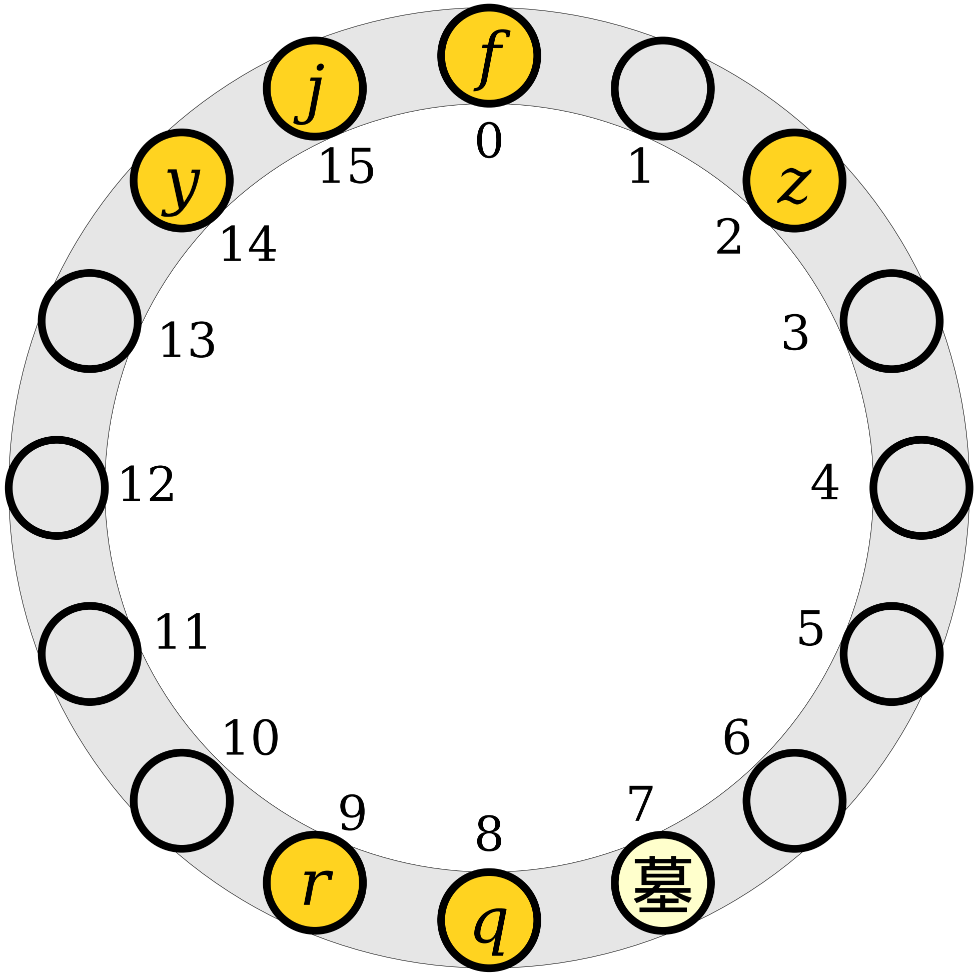
Linear Probing

- Having too many tombstones in a table can slow down lookups, since we have to scan past them.
- Tombstones should be overwritten when new elements are inserted.



Linear Probing

- Be careful, though, to make sure you don't allow for duplicates in your table.



★ Linear Probing at a Glance ★

- To check if an element exists in the table:

  Compute the hash code of the element.

  Jump to that location in the table.

  Scan forward – wrapping around if necessary – until the item or an empty slot is found.

- To insert an element into the table:

  If the item already exists, do nothing.

  Otherwise, jump to the slot given by the hash code of the element. Walk forward – wrapping around if necessary – until a blank spot or tombstone slot is found. Then, put the item there.

- To remove an element from the table:

  Jump to the slot given by the hash code of the element.

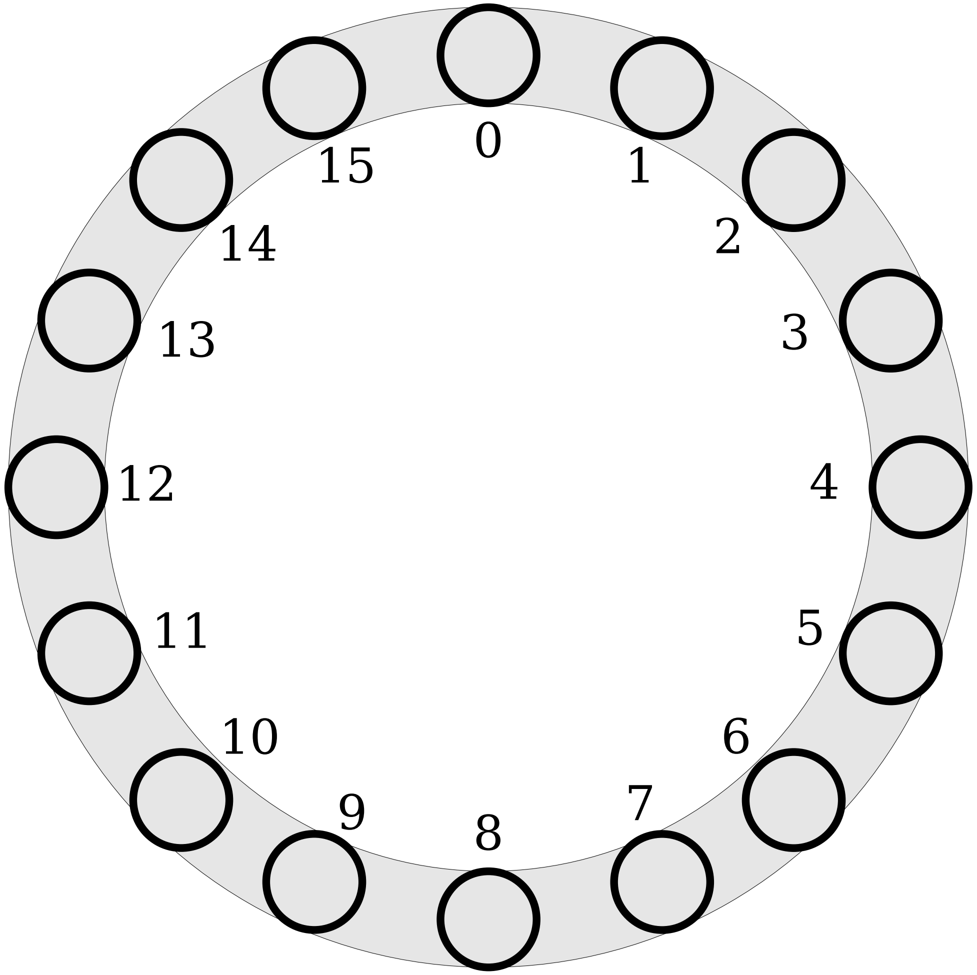
  Walk forward – wrapping around if necessary – until the item or an empty slot is found. If the item is found, replace it with a tombstone.

How Fast is Linear Probing?

- ***Recall:*** The ***load factor*** of a hash table, denoted ***α***, is the ratio of the number of items in the table to the number of slots.
- ***Fact:*** For any fixed value $\alpha < 1$, the expected cost of a lookup in a linear probing table is O(1), assuming you have a good hash function (and you rehash when the table gets too full).
- This is the same big-O cost as a chained hash table, though with a totally different strategy!

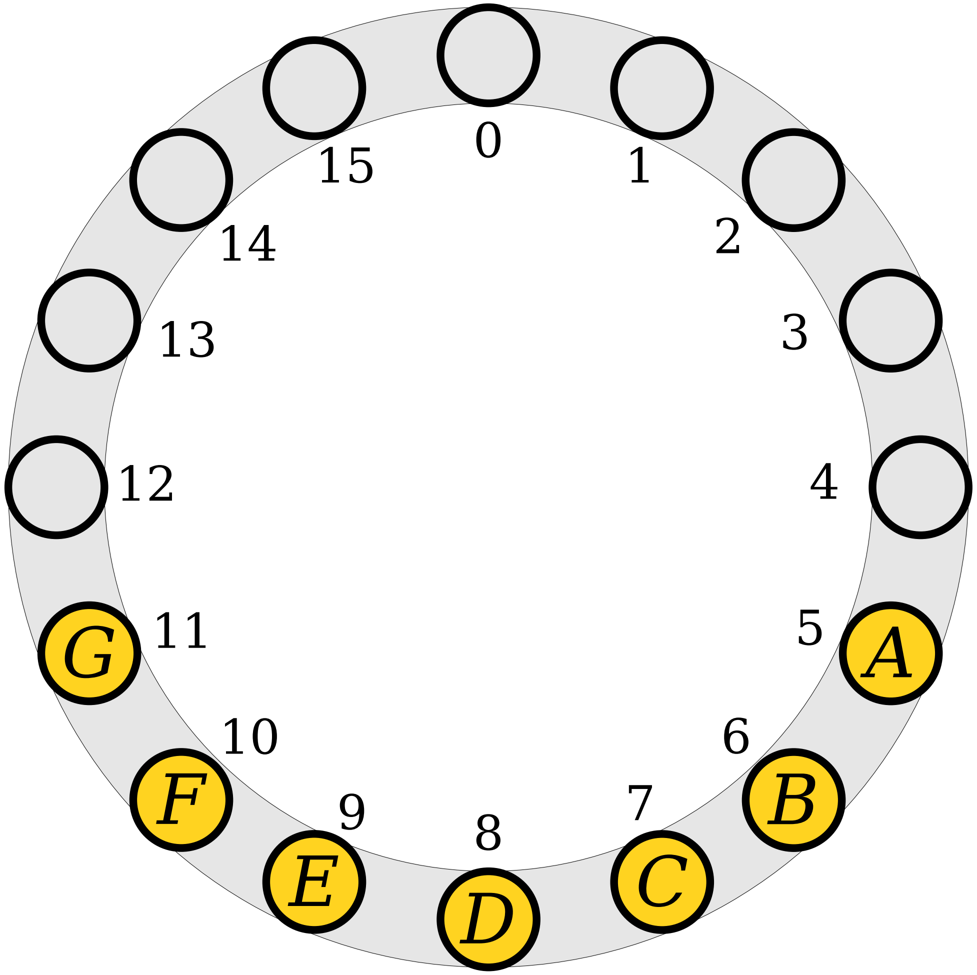
Try It Yourself!

- Insert the following values into this table.
  - *A* (hash code 5)
  - *B* (hash code 5)
  - *C* (hash code 5)
  - *D* (hash code 8)
  - *E* (hash code 7)
  - *F* (hash code 6)
  - *G* (hash code 5)



Try It Yourself!

- Insert the following values into this table.
  - A (hash code 5)
  - B (hash code 5)
  - C (hash code 5)
  - D (hash code 8)
  - E (hash code 7)
  - F (hash code 6)
  - G (hash code 5)



Time-Out for Announcements!

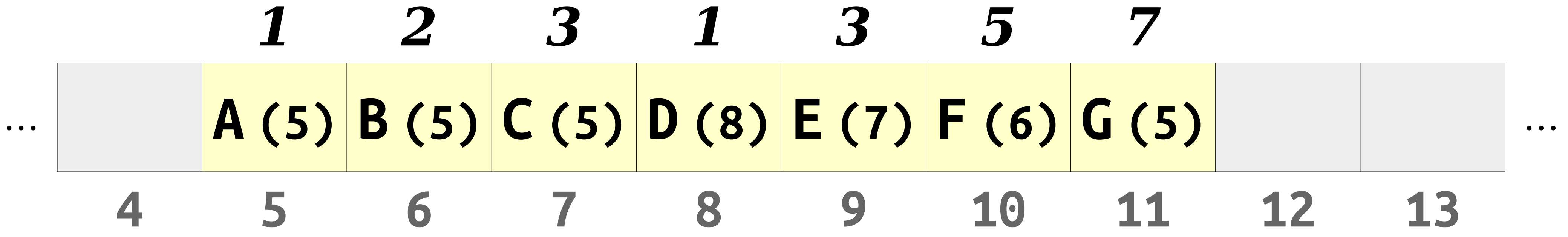
Assignment 6

- Assignment 5 was due today at 1:00PM.
  - Want to use your late days? You can extend the deadline by 24 or 48 hours.
- Assignment 6 (***The Great Stanford Hash-Off***) goes out today. It's due next Friday.
  - Implement the hashing strategies from today!
  - See how fast these approaches are and how they compare against chained hashing!
- As always, come talk to us if you have any questions! That's what we're here for.
- YEAH Hours will be recorded and posted today. Due to low attendance we won't be holding them in-person this time. If you'd like us to bring those back, let us know!

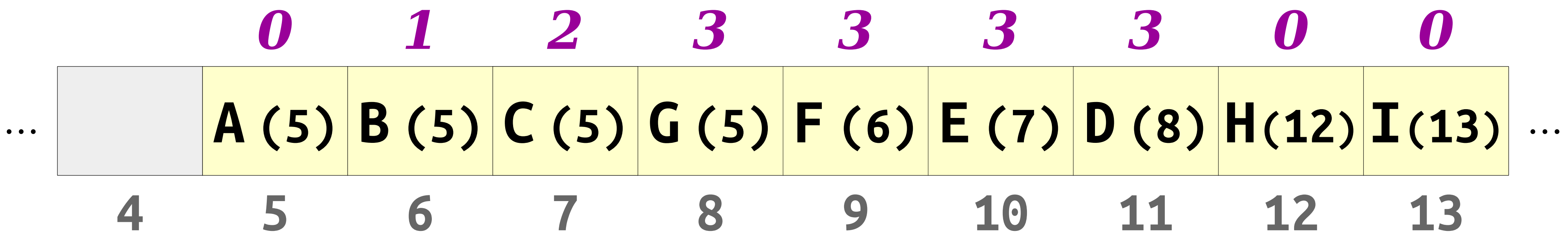
Back to CS106B…

A Question of Fairness

- Suppose we look up each of these elements. How many slots would we need to look at to find each of them?
- There's a large *variance* in how long it's going to take to find things.
- How can we fix this?

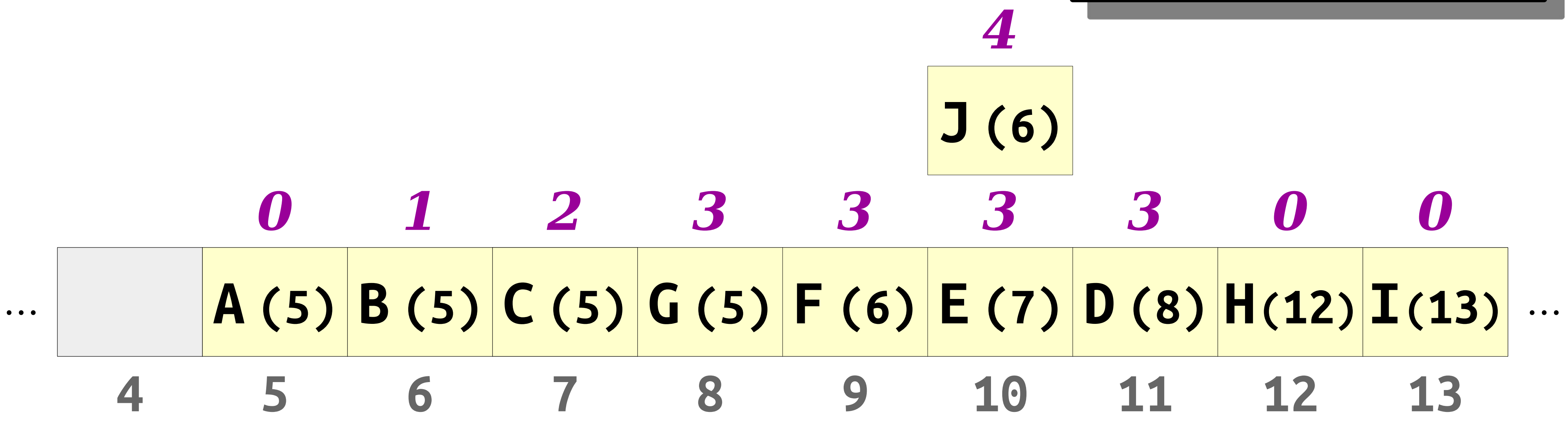


- ***Robin Hood hashing*** is a slight modification to linear probing.
- When we insert an element, if the element we're inserting is further from home than the current element, we displace that element to make room for the new one.

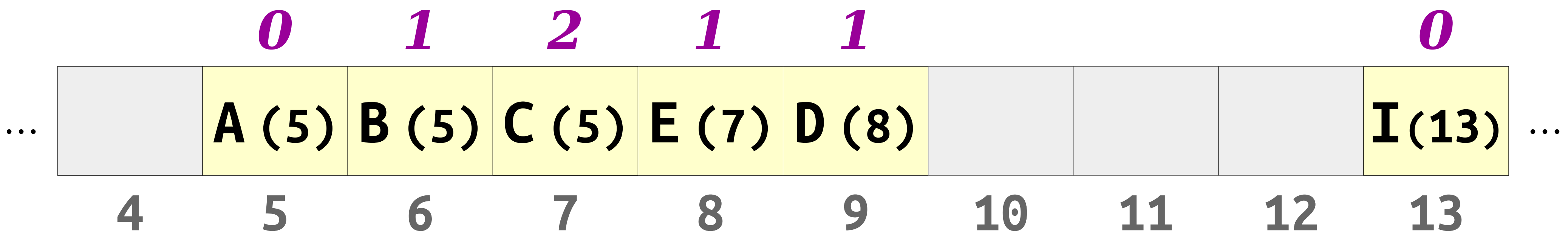


- ***Neat trick:*** We can make unsuccessful lookups in a Robin Hood hashing table faster than in a linear probing table.
- ***Idea:*** Compare the distances of the item to insert and the item being looked up.

If **J** were in this table, it would have displaced the **E**. So **J** can't be in the table!



- ***Neat trick:*** Robin Hood hashing doesn't need tombstones.
- We can use a technique called ***backward-shift deletion*** instead.



★ Robin Hood Hashing at a Glance ★

- To check if an element exists in the table:

  Jump to the spot in the table given by the element's hash code.

  Scan forward – wrapping around if necessary – keeping track of how many steps you've taken. Stop when you find the item, you find a blank slot, or you find a filled slot closer to home than the number of steps you've taken.

- To insert an element into the table:

  If the element is already in the table, do nothing.

  Jump to the table slot given by the element's hash code. Scan forward – wrapping around if necessary – keeping track of the number of steps taken. If you find an empty slot, place the element there. Otherwise, if the current slot is full and closer to home than the element you're inserting, place the item to insert there, displacing the element that was at that spot, and continue the insertion as if you were inserting the displaced element.

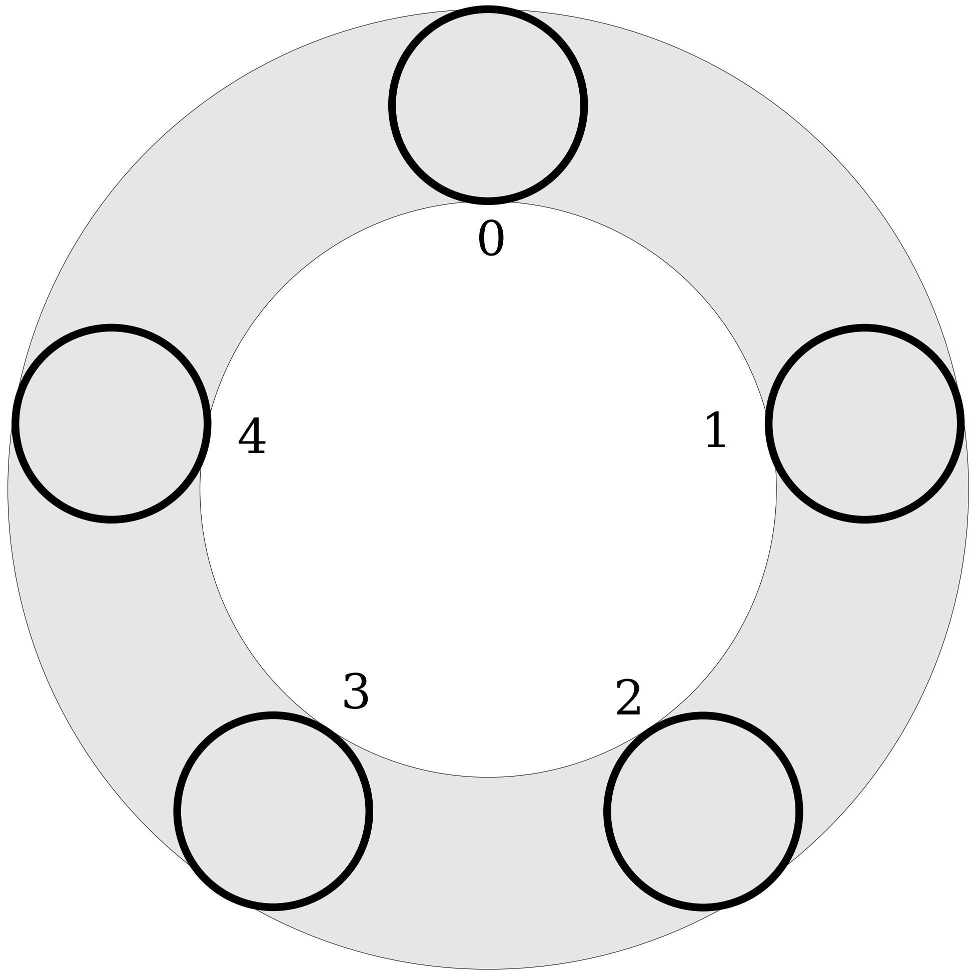
- To remove an element from the table:

  Jump to the slot given by the hash code of the element.

  Walk forward – wrapping around if necessary – until the item or an empty slot is found. If the item is found, remove it. Then, keep moving forward – wrapping around as necessary – moving elements backward in the table one slot until an empty slot or an item in its home position is found.

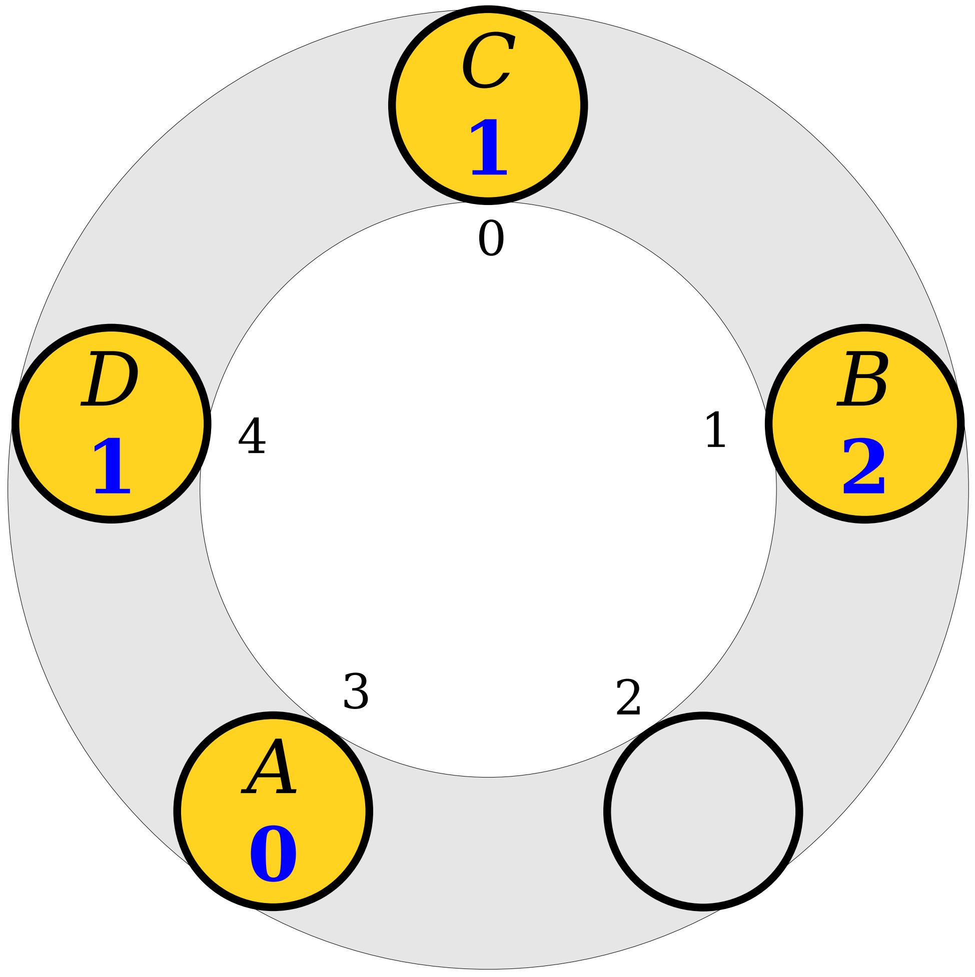
Try It Yourself!

- Draw what happens if we insert the following values into this Robin Hood hash table.
  - *A* (hash code 3)
  - *B* (hash code 4)
  - *C* (hash code 4)
  - *D* (hash code 3)



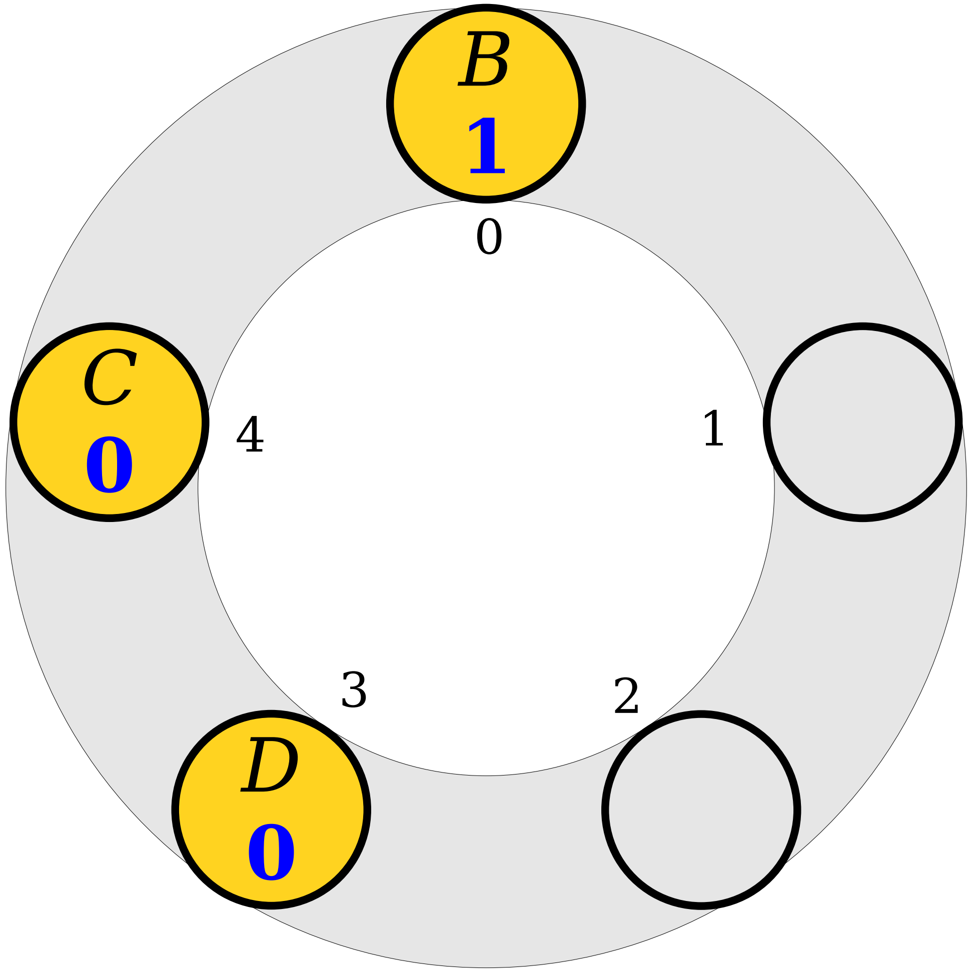
Try It Yourself!

- Draw what happens if we insert the following values into this Robin Hood hash table.
  - *A* (hash code 3)
  - *B* (hash code 4)
  - *C* (hash code 4)
  - *D* (hash code 3)
- What happens if we delete *A*?



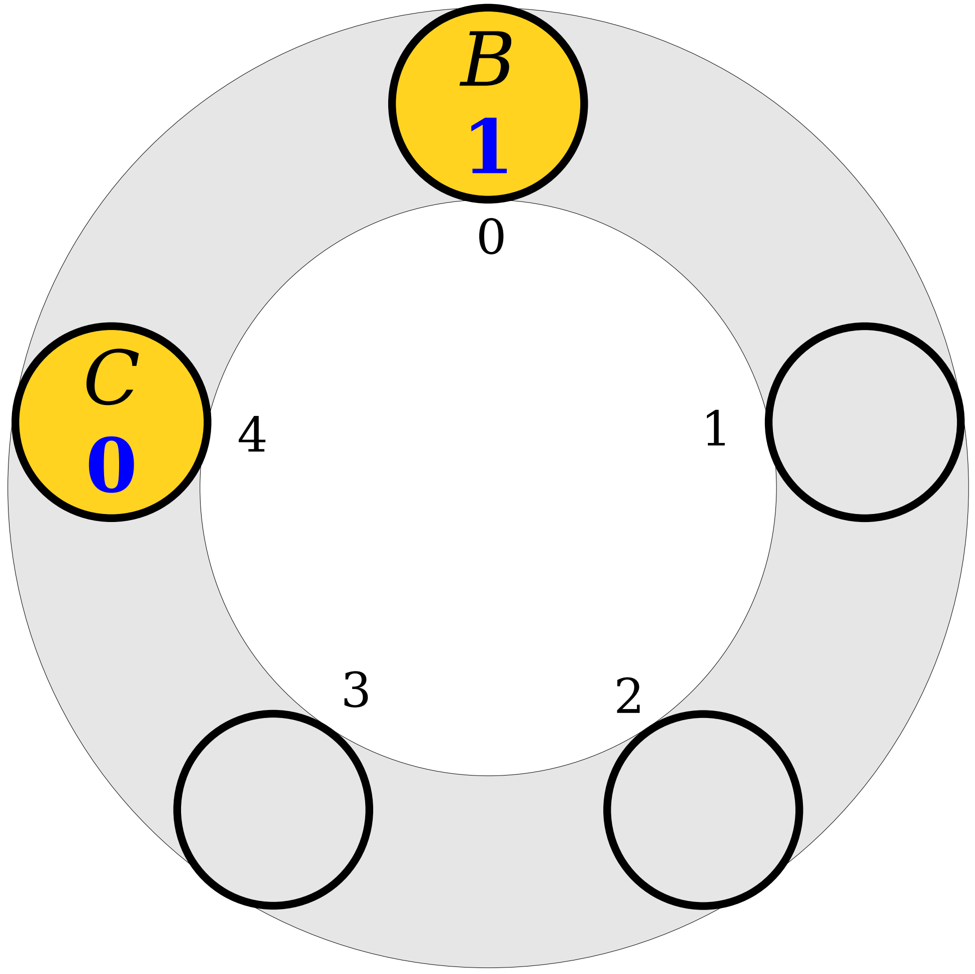
Try It Yourself!

- Draw what happens if we insert the following values into this Robin Hood hash table.
  - *A* (hash code 3)
  - *B* (hash code 4)
  - *C* (hash code 4)
  - *D* (hash code 3)
- What happens if we delete *A*?
- What happens if we now delete *D*?



Try It Yourself!

- Draw what happens if we insert the following values into this Robin Hood hash table.
  - *A* (hash code 3)
  - *B* (hash code 4)
  - *C* (hash code 4)
  - *D* (hash code 3)
- What happens if we delete *A*?
- What happens if we now delete *D*?



Robin Hood Hashing

- Like linear probing, with a good hash function, the expected cost of a lookup in a Robin Hood hash table is O(1).
  - (Assuming you have a fixed load factor α and rehash the table when you get too full.)
- Robin Hood hashing requires a bit of extra work compared to linear probing for the distance bookkeeping.
- However, the speedups from cutting off searches early and for not having tombstones are significant at high load factors.
- ***Optional:*** Code up Robin Hood hashing and compare it against linear probing.

Your Action Items

- ***Start Assignment 6***
  - You know the drill! Slow and steady progress is the name of the game here.
  - Aim to complete Enumerations Warmup and Linear Probing Warmup tonight, then start working on Implementing Linear Probing tomorrow.

Next Time

- ***Linked Lists***
  - A different way to store a sequence.
- ***Recursive Data Types***
  - Data types defined in terms of themselves.