Question 1. Suppose I want to store information about UCI students in a map-like data structure. I could declare `std::vector<Student> students` with a capacity 100 million and store each student in the spot for their ID #. What could go wrong? Are there any advantages to this idea?

What's the big idea?

- How do we decide the *table size* (array size)?
- How do we decide where to put a data?
- What if two data elements need to share a space?

What is the "load factor" of a hash table? Why is it important?

Question 2. What constitutes a good hash function?

Example 1:

- Table size 13
- Compression function: $h(k) = k\%13$
- Insert 28, 18, 10, 25, 41

0	1	2	3	4	5	6	7	8	9	10	11	12

Question 3. What does "chaining" refer to in collision resolution? Is it a good idea?

Example 2:

- Table size 13
- Compression function: $h(k) = k\%13$
- Insert 28, 18, 10, 25, 41, 36, 38, 54, 12, 90

0	1	2	3	4	5	6	7	8	9	10	11	12

Question 4. How good is chaining? Does it solve the problem well?

Question 5. Can a word/name be a number? Assume the `std::string` contains only letters A-Z for this question.

Question 6. How does quadratic probing work, if our initial hash function is $h(k) = k\%m$, our table size $m = 13$, and we insert 25, 48, 8, 17, 42, 38, 64?

0	1	2	3	4	5	6	7	8	9	10	11	12

Question 7. How does removing from a hash table work? Suppose I have the table below, we are using quadratic probing as above, and I want to remove 38. What happens if I search for 64 afterward?

0	1	2	3	4	5	6	7	8	9	10	11	12
38		64	42	17				8	48			25

Within the context of data structures, we use hash functions to convert a key to an array index.
Question 8. Suppose our keys are positive integers. What is a common hash function and why?

Question 9. Suppose our keys are *floating point* numbers. What are two common hash functions and why?

Hash functions for use in a hash table have three requirements; hash functions for cryptography-related purposes have different needed properties. For a hash table, the three needed ones are:

- Consistent
- Efficient to computer
- Output provides a uniform distribution for the set of keys.

Question 10. How might you test a hash function before considering it for inclusion in a hash table?

When using **Polynomial Evaluation Functions**, we choose a non-zero value of $a \neq 1$ and use $h(k) = x_1a^{d-1} + x_2a^{d-2} + \ldots + x_{d-1}a + x_d$

This can be written as: $h(k) = x_d + a(x_{d-1} + a(x_{d-2} + \ldots + a(x_3 + a(x_2 + ax_1))\ldots))$
Question 11. Why is this called a polynomial evaluation function? What role within the polynomial do tuple pieces serve?

Cuckoo Hash Tables

So far, we have seen hash tables and many positives aspects of them. Today's lecture starts with the idea that if one hash table is good, two must be even better! So let's look into having a dictionary interface (insert, delete, search) but, instead of having one table of linked lists (chaining), or one table with open addressing (linear and quadratic probing), we will have:

- Two tables, which we will call H_0 and H_1
- Two different hash functions
  - $h_0(k)$ will be used for H_0
  - $h_1(k)$ will be used for H_1

Question 12. How do we search to determine if a key is in our table? How long does this take?

Question 13. How do we delete a key if it is in our table? How long does this take?

Question 14. How do we insert into the table?

Example: Insert the following into the dictionary that is being implemented via Cuckoo Hashing. Use $h_0(k) = k\%11$ and $h_1(k) = (k/11)\%11$

The key sequence is: 12, 26, 92, 23, 28, 94, 15

0	1	2	3	4	5	6	7	8	9	10

Theorem: Let G be an undirected connected graph. We can direct the edges of G such that no vertex has more than one incoming edge if and only if G has at most one cycle.

Proof: (to be supplied in lecture)

Question 15. What does this theorem tell us about cuckoo hashing?