

Hashing

- Order of data has *meaning*
 - ArrayList/LinkedList: Position in ADT has meaning
 - Stack/Queue: Order of arrival/exit

Consider:

```
ArrayList<Integer> L;  
L.get(0);  
L.get(1);
```

These are different objects, even if the value is the same!

- These are called **Ordered** Data Structures
 - The common thread? Searching is **slow**

- Another way to store data
 - Elements are stored in a **Hash Table** (usually an Array)
 - The element's position in the table is dependent on the **hash function**

Example Hash Function (Integers)

$$h(x) = x \% \text{table.length}$$

Example

Given a hash table of size 10, insert the following numbers: 23, 31, 57, 26, 90, 18.

$$h(x) = x \% 10$$

Solution

$23 \% 10 = 3$, so 23 will be stored in index 3. If we do this for all numbers, we get:

90, 31, \emptyset , 23, \emptyset , \emptyset , 26, 57, 18, \emptyset

Inserting into a hash table is **fast** (assuming a good and fast hashing function)

- Does 42 exist in the table?
 - $h(42) = 42 \% 10 = 2$
 - `table[2] == null`
Nope!
- Does 23 Exist in the table?
 - $h(23) = 23 \% 10 = 3$
 - `table[3] == 23`
Yes!

- What happens if we insert 13 into the list?
 - $h(13) = 13 \% 10 = 3$
 - `table[3]` Contains 23
 - This is called Collision
- Perfect Hashing Function
 - A hash function that gives no collisions
- Perfect hashing functions are nice, imperfect ones can still be very fast!
- The more you know about your data, the more likely you can get a perfect hash function
 - This may be possible, but impractical

Impractical Hashing Functions

- You have 200 employees. How could you hash the employee records?
- SSN is unique per employee, so you could use a SSN as an index into an array
- Problem: There are 999,999,999 possible SSNs, and you are planning on using 2000 elements of the array. That's very inefficient!

Common Hashing Functions

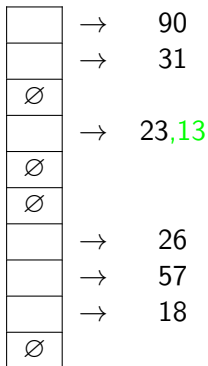
- Hashing functions usually rely on a numerical key
 - There are ways to turn any object into a number, if you try hard enough!
- After objects are represented by a number, common ways to store in a table include:
 - Mod by the length of the list, and the result is the index
 - Radix transformation: transform into another base, then mod

Handling Collisions

- There are two common ways to deal with collisions:
 - Chaining
 - Open Addressing

Chaining

Instead of storing individual values in the table, it is a reference to a list of items with the same hash key



- If a collision occurs, look for another available spot
- There are different ways to do this
 - Linear Probing
 - Quadratic Probing

Simply walk along the table to the next available spot. Since index 3 contained 23, we try index 4, which is empty.

90, 31, \emptyset , 23, 13, \emptyset , 26, 57, 18, \emptyset

If you reach the end of the table, wrap back around to the beginning

Instead of incrementing the index by one, try:

$$h(x) + (-1)^{i-1} \left(\frac{i-1}{2} \right)^2$$

In other words, try:

$$h(x), h(x) + 1, h(x) - 1, h(x) + 4, h(x) - 4, h(x) + 9, h(x) - 9, \dots$$

This may be preferable to avoid clumps (depending on your data)

Table Size

- The size of the table is important
 - Too large: space wasted
 - Too small: Lots of collisions, very slow
- Rule of Thumb: Have your table be 1.5x the size of your data
- Dynamic Resizing
 - If your data size is unknown, start with a size
 - When the table is a certain percent full, increase the size
 - **Note: resize is slow** since you must rehash all elements
 - Waiting too long to resize may give bad performance

Characteristics of Hash-Based ADTs

- The Good

- With a good hash function, search and insert are very fast
- Great if your elements don't need an ordering
- Great if your data size is huge

- The Bad

- No ordering: If you need an ordered data set, this isn't for you
- Frequent resizing is slow, as all elements must be rehashed