Section 5: Hashing & Sorting

0. Hash... Browns?

For the following scenario, insert the following elements in this order: 7, 9, 48, 8, 37, 57 into the table on the left. For each table, TableSize = 10, and you should use the primary hash function *h(k)* = *k*. If an item cannot be inserted into the table, please indicate this and continue inserting the remaining values.

Once you have inserted the elements, delete the elements in this order: 37, 7, 57, and show what the hash table will look like after this deletion on the right table.

a) Linear Probing (insertion)

0	8
1	37
2	57
3	
4	
5	
6	
7	7
8	48
9	9

b) Linear Probing (deletion)

0	8
1	X
2	X
3	
4	
5	
6	
7	X
8	48
9	9

c) Separate chaining hash table - Use a linked list for each bucket. Order elements within buckets in any way you wish.

0	
1	
2	
3	
4	
5	
6	
7	57→37→7
8	8→48
9	9

1. Let's Hash Out Hashing

a) Describe double hashing.

Solution:

The first hash function determines the original location where we should try to place the item. If there is a collision, then the second hash function is used to determine the probing step distance as 1*h2(key), 2*h2(key), 3*h2(key) etc. away from the original location.

b) Compare open hashing and separate chaining.

Solution:

Open Hashing

- Deals with collisions by moving element to a different index
- Uses less memory
- Linear probing: easiest, but has primary clustering, need to resize a lot
- Quadratic probing: secondary clustering, will find a spot if $\lambda < ½$
- Double hashing: low chance of clustering, but need another hash function

Separate Chaining

- Deals with collisions by putting all the elements in a "bucket" at that index
- Easier to implement
- More memory because needs "bucket" data structure
- Average runtime for insert/delete/find: $O(1 + \lambda)$
    - Best: O(1)
    - Worst: O(n)