



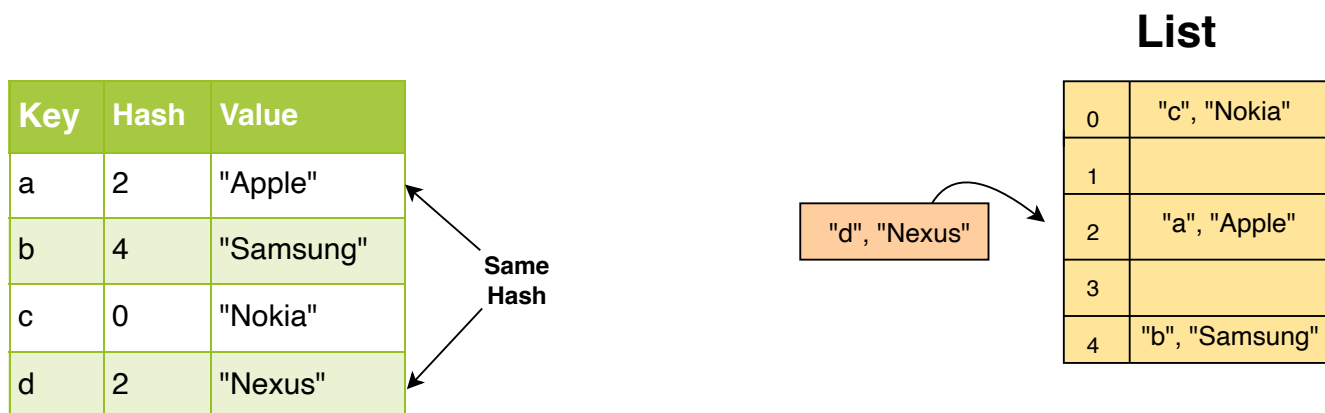
Collisions in Hash Tables

This lesson is about how collisions occur in hashing and the common strategies used in resolving these collisions.

We'll cover the following

- Strategies to Handle Collisions
 - Linear Probing
 - Example
 - 2. Chaining
 - 3. Resizing the List

When you map large keys into a small range of numbers from 0-N, where N is the size of the list, there is a huge possibility that two different keys may return the same index. This phenomenon is called **collision**.



Strategies to Handle Collisions#



There are several ways to work around collisions in the list.



common strategies are:

- *Linear Probing*
- *Chaining*
- *Resizing the list*

Linear Probing#

This strategy suggests that if our hash function returns an index that is already filled, move to the next index. This increment can be based on a fixed offset value to an already computed index. If that index is also filled, traverse further until a free spot is found.

One drawback of using this strategy is that if we don't pick an offset wisely, we can end up back where we started and, hence, miss out on so many possible positions in the list.

Example#

Let's say the size of our list is **20**. We pass a key to the hash function which takes the modular and returns **2**.

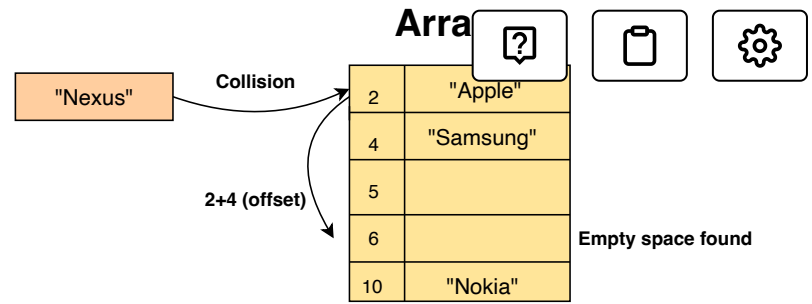
If the second position is already filled, we jump to another location based on the **offset** value. Let's say this value is **4**. Now we reach the sixth position. If this is also occupied, we repeat the process and move to the tenth position and so on.

Here's an illustration which will make things clearer:



Key	Hash	Value
2	$2\%20 = 2$	"Apple"
4	$4\%20 = 4$	"Samsung"
30	$30\%20 = 10$	"Nokia"
42	$42\%20 = 2$	"Nexus"

Same Hash

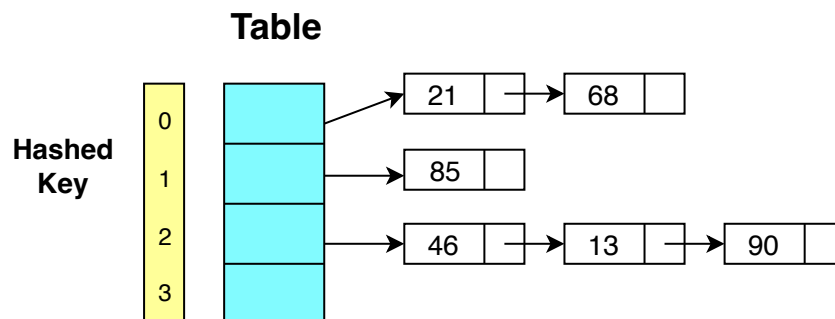


2. Chaining#

In the chaining strategy, each slot of our hash table holds a pointer to another data structure such as a linked list or a tree. Every entry at that index will be inserted into the linked list for that index.

As you can see, chaining allows us to hash multiple key-value pairs at the same index in constant time (insert at head for linked lists).

This strategy greatly increases performance, but it is costly in terms of space.






Collisions Handled by Forming a Linked-List at each index

3. Resizing the List#

Another way to reduce collisions is to resize the list. We can set a threshold and once it is crossed, we can create a new table which is double the size of the original. All we have to do then is to copy the elements from the previous table.




Resizing the list significantly reduces collisions, but the function is    costly. Therefore, we need to be careful about the threshold we set. A typical convention is to set the threshold at **0.6**, which means that when 60% of the table is filled, the resize operation needs to take place.

Another factor to keep in mind is the content of the hash table. The stored records might be concentrated in one region, leaving the rest of the list empty. However, this behavior will not be picked up by the resize function and you will end up resizing inappropriately.

Some other strategies to handle collisions include *quadratic probing*, *bucket method*, *random probing*, and *key rehashing*. We must use a strategy that best suits our hashing algorithm and the size of the data that we plan to store.

With our theoretical fundamentals in place, we can move on to the building our first hash table in Python.

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. [See how](#) 



 Back

Next 

The Hash Function

Building a Hash Table from Scratch



Mark as Completed



Report an Issue



