Hashtable

- If we store the pair at $index = hash(key)\ \%\ len(table)$, then the location of the pair is immediately known.
- A **hashtable** is an implementation of the Map that uses the hash for a key to compute an index into an array where the corresponding key-value pair will be stored.

Taking Care of Collision

- When there are not so many spots in a Hashtable, it is easy to have two key-value pairs put into the same spot. When there are more than one items put into the same spot, we have a collision.

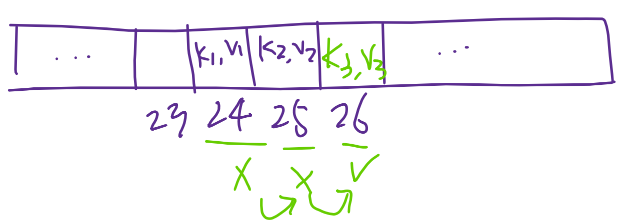
1. When there are p items and n spots, $n \geq p$, what is the probability that all items are put into different spots?
   - $P[\text{all items are put into different spots}] = \frac{N[\text{all items are put into different spots}]}{N[p \text{ items are put into } n \text{ spots}]}$
   - $N[p \text{ items are put into } n \text{ spots}] = n^p$, because each item has n choices and there are p items
   - $N[\text{all items are put into different spots}] = n \times (n-1) \times (n-2) \times \ldots \times (n-p+1) = \frac{n!}{(n-p)!} = P(n,p)$, because the first item has n choices, the second item has $n-1$ choices, and so on.
   - $P[\text{all items are put into different spots}] = \frac{P(n,p)}{n^p}$

2. What is the possibility that all 50 students have different birthdays?
   - $\frac{P(366,50)}{366^{50}} \approx 0.03$

Let's look at some designs to take care of collisions.

- One way to take care of collision is to put the key-value pair to the next available spot in the Hashtable.
  - For example, if "200": "*banana*" is mapped to index 24 in a Hashtable with 100 spots, but "100": "*apple*" is already in this index, we put "200": "*banana*" to index 25 instead. However, if index 25 is also taken, we put "200": "*banana*" to index 26 instead.

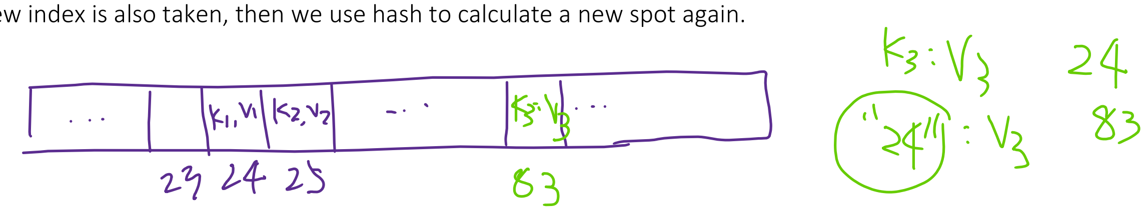
$K_3 : V_3$ 24



  - When we call $Hashtable["200"]$, we will check index 24 at first, and we won't find the item, then we need to linear search from index 24 and on. When there are not that many collisions, we still expect only constant number of additional checks to find "200": "*banana*".
  - The disadvantage of this design is very obvious, one collision may trigger another collision. In the end, it is possible that none of the key-value pairs are stored at the expected spot.

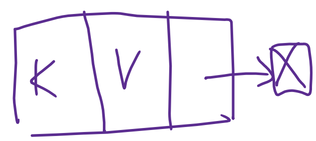
- Another way to take care of collision is to "double hash".

- For example, if $"200": "banana"$ is mapped to index 24 in a Hashtable with 100 spots, but $"100": "apple"$ is already in this index, we call $hash("24")\ \%\ 100$ to find a new spot for $"200": "banana"$. Again, if the new index is also taken, then we use hash to calculate a new spot again.

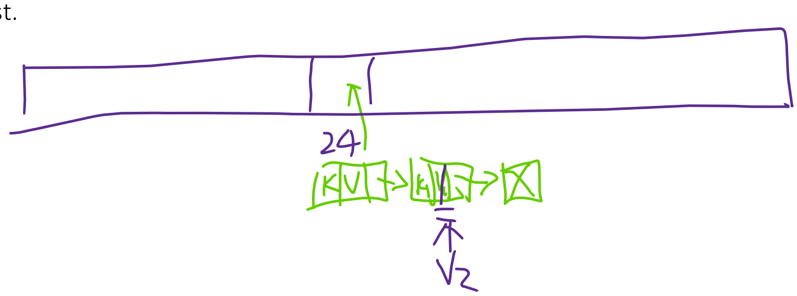


  - ➢ When we call $Hashtable["200"]$, we will check index 24 at first, and we won't find the item, then we need to while loop to hash again and again until we find $"200": "banana"$. When there are not that many collisions, we still expect only constant number of iterations to find $"200": "banana"$.
  - ➢ Similar to the first design, one collision may trigger another collision in the design.

- We will use the following design in this class (and in the assignment). We take care of collision using Singly LinkedList.
  - We wrap each pair of key-value into a node that point to None.

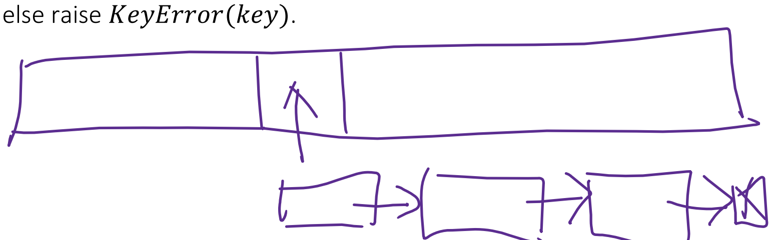


  - __setitem__(key, value):
    - ➢ We still call $hash(key)\ \%\ len(table)$ to calculate the index for this key-value pair. If the spot in empty, we add this node to this index.
    - ➢ If the spot is taken, we look at the LinkedList in this spot. If we find a node in this list with the same key, update its old value to the new value; or else we append a new node with key-value to the tail of this list.



index
K : V : 24
K1 : V1 : 24
K1 : V2 24

  - __getitem__(key):
    - ➢ We still call $hash(key)\ \%\ len(table)$ to calculate the index for this key-value pair.
    - ➢ We search in the LinkedList that's stored in this index. If we find the node with given key, return its value; or else raise $KeyError(key)$.



  - __contains__(key):
    - ➢ Call $hash(key)\ \%\ len(table)$ to calculate the index, and search in the LinkedList that's stored in this index. Return True if key is in that list; or else return False.

- How to efficiently implement __iter__() in a Hashtable with n spots and k key-value pairs?
  - If we don't know what the keys in this Hashtable are, we can only scan the table to find all LinkedLists. The time complexity of this operation is $\Theta(n)$.

  - We can add a list (the Python list object, or an ArrayList) that contains all keys as an attribute of the Hashtable. Then, we only need to yield the keys in this list for __iter__() method.
    - This list of keys takes only $\Theta(k)$ extra memory, but it reduces the time complexity of the iterator method from $\Theta(n)$ to $\Theta(k)$.
    - Whenever we __setitem__(.) with a new key, we append the new key to the key list.