## Hash Tables

→ most practical data structure for implementing a map
→ intuitively, a map M supports abstraction of using keys as indices with a syntax such as M[k]
→ ideally: keys would be well distributed from 0 to N-1 by a hash function
→ in practice, there could be a few distinct keys mapped to same index
→ we conceptualize a bucket array, where each index (bucket) may manage a collection of items sent to a specific hash by a hash function

## Hash Functions

→ goal of a hash function is to map each key k to an integer in range [0, N-1], N being the capacity of the bucket array for a hash table
→ we store the item (key, value) in bucket A[h(key)]
→ If 2 or more keys in bucket w/ same hash value: a collision has occurred
→ common to view evaluation of hash function, h(k), consisting of 2 portions:
1.) hash code that maps key to an integer
2.) compression function that maps hash code to integer within [0, N-1]

## Collision Handling

→ existence of collisions prevent us from simply inserting new item (k, v) directly into bucket A[h(k)]
→ Separate Chaining:
→ simple way of having each bucket A[j] store its own container holding items (k, v) such that h(k) = j
→ natural choice for secondary container is small map instance implemented using a list
→ Open Addressing: if space is at a premium, rather than using auxiliary data structure
→ we can use approach of always storing item directly in a table slot
→ Linear Probing:
→ If we try to insert item (k, v) into a bucket A[j] that is already occupied, then next we try A[(j+2) mod N] and so on until we find empty bucket
→ this strategy requires changing implementation for searching for an existing key