**Hash Table**

A data structure allowing finding a large collection of data.

* Hash Table provides O(1) average performance on search/insert/deletion
* Stores data in a very large array
  + Typically 2x as many elements as the # of data elements to be stored
* A hash function:
  + input: **key** - relates to a **value** (that gets inserted with the key)
  + output: **index** of the array

For example, the hash function is implemented as **h(k) = k % (m+1)** (stores the value at the index that corresponds to the modulus of k)

\*\*What if we try to insert 2 elements that have the same modulus? 🡪 **Collision!**

Diagram, schematic

Description automatically generated **Solving Collision**

1. **Hashing with chaining**
   1. Each hash table entry contains a pointer to a **linked list of keys that map to the same index**
   2. Therefore, the worst-case complexity for search is O(n) (traverse the linked list)
2. **Avoiding collision**
   1. Expand such that the average length of each list is 1 (distributed)
      1. Use more spaces (2x array elements)
      2. When collision occurs, store at the next empty position. Add metadata to signal that we are doing this (Google algorithm)
   2. Use smarter algorithm 🡪 usually involve multiply a large prime number
      1. e.g. h(k) = k \* 31 % m

\*\* Which way is faster? Expanding! Although it requires more metadata and more sophisticated implementations, the implementation is much faster, due to **cache** instead of **memory** (required for linked lists)

**String keys**

ASCII values for each letter? (Good for small number of letters only)

**Java** chooses having large hash table over having any collisions (for string s[n])

