ADTs
An ADT is a set of defined instructions that must be satisfied by any data structure that implements the ADT

Benefits of ADTs
Information Hiding: Abstracts implementation details from user, the same operations are always available
Re-usability: Implementation can easily be reused accross different programs

Dictionary ADT:
- **Methods:** get(key) - return data associated with given key, or null if key doesnt exist
  put(key, data) - inserts key and data into dictionary, or error if dictionary already contains the key
  remove(key) - removes record of given key or ERROR if key DNE
- A dictionary can be implemented with a Hash Table, BST, Linked List, etc

Hash tables are used because they have perform operations in O(1) average time
These operations can be O(1) because you can jump directly to a keys specific index using the hash function.

```
get(key) {
    return dictionary[h(key)].value
```

This will hash the key and know the index it is stored at in O(1) without searching others

The reason this is average time and not worst case is because of collisions...

Hash Tables

Hashing is a method of storing data using a hash function that takes a key value pair and assigns it to an index in a Hash Table

Lets use the example of a database that stores student data and student ID numbers.
Key = 9 digit ID
Value = Students data

These IDs will be hashed an placed in the table bringing their data with them. But how is this done

Hash Function Basic Example

Dictionary (not yet implemented with hash table)
251448535 —> Jack, CS, Sophomore

Lets say we have 50k students, we need a hash table with 50,000 slots so lets define a hash function to map each ID to a spot in the table.

hash(n) = n % 50,000 —> mod by number of slot ensures it will be an integer ≤ table size
hash(251448535) = 48,535 —> This ID and its data is hashed to slot 48,535

So now: put(key, value)
    dictionary[48,535] = (251448535, "Jack, CS, Sophomore")
And then: get(key)
    dictionary(hash(key)).Value

Example 2, Polynomial String Hashing:

Lets say instead of the key being ID it was first name (for example purposes)

Jack → CS, Sophomore

Define polynomial hash function : $h(k) = (C_0 + C_1 K + C_2 K^2 + \cdots + C_{k-1} K^{k-1}) \bmod M$

C_i = ASCII value of letter i in input string
K = Some small arbitrary prime number like 31 or 37
M = size of table

ASCII for example String: Jack = (74, 65, 67, 75)

$h(\text{"JACK"}) = (74 + 65 \cdot 31 + 67 \cdot 31^2 + 75 \cdot 31^3) \bmod 50{,}000$
$= 2300801 \bmod 50{,}000$
$= 801$

→ Multiplying $c_0, c_1 \ldots$ by different powers of 31 makes JACK different than any other 4 letter combination of these letters.

→ Primes are used because less factors = less accidental patterns form when modding

put("JACK", data)
will internally: Dictionary[801] = ("JACK", record)

Horner's Rule:

ints in Java are fixed to a size of 32 bits, meaning extremely large integers will cause integer overflow

To solve this, take mod after each step: $h = ((\cdots(C_0 K + C_1) K + C_2) K + \cdots + C_{n-1}) \bmod m$

In Java:
```
for (int i=0; i<key.length(); i++)
    h = (h * K + key.CharAt(i)) % m;
```

Collisions

A good hash code will minimize the time hash(key₁) = hash(key₂) but not eliminate it. This called a collision.

Separate Chaining:

Each slot in the table holds a LinkedList of all the keys in that hash. When a collision occurs, append to chain
Pros: Table can handle many collisions, easy to implement
Cons: Extra pointers in memory, increased lookup time (O(n))

Open Addressing:

Each key gets it's own index in the table. When a collision occurs, you probe for an open spot.

Linear Probing: Move to the next slot until you find an empty one: $(h_i(k) + i) \bmod M$
Pros: Simple
Cons: may take longer to find an open slot due to clustering

Quadratic Probing: Jump up by squares: $h_i(k) = (h(k) + i^2) \bmod M$
Pros: Reduces clustering
Cons: Can miss open slots

Double Hashing: $h_i(k) = (h_1(k) + i \times h_2(k)) \bmod M$
Pros: Avoids clustering
Cons: More computation