Hash map

* Stores an array of “buckets” which are just positions that can hold Entries (key value pair objects)
* How it works is:

1. The key is converted to a numeric value
2. The numeric value is divided by the size of the array using mod
3. The answer is used to find out which position in the array to store the entry
4. The entry is placed there

* How every collision can happen, the best way I have found to deal with this is linear probing: because:

1. Linear probing works well for small tables, because of better local memory caching
2. Fast edge look ups are crucial (we will most likely use operation to check if two pixels are connected) and liner probing has low overhead compared to other methods compared to this.
3. Since we using images, memory efficiency is a big concern and linear probing has no extra pointers

* A concern Is clustering were elements group together, in certain regions in the hash table, leading to performance degradation.

.

However, our implementation will however optimize liner probing to improve the HashMap data structure by:

1. Using a low load factor because since adjacency maps are small resizing will not be expensive, and this will help avoid clustering.
2. Using good hash functions murmurhash or multiply shift, as these hashes work well for coordinates.
3. Using robin hood probing, to help reduce worst case hashing
4. Since our water analysing may involve dynamic updates we will need efficient lookups.
5. We will use lower load factors (0.5, resize when half full), because they work well with linear probing, and since our adjacency maps is likely to be small, this will help avoid clustering

Our hashing function: Two prime mix

* Uses this because pixels have spatial patterns that simple hashes amplify.
* 2 prime number multiplier breaks up grid like patterns
* XOR shifting provides final mixing

Our probing/ hashing algorithm: Robin Hood Hashing

* One simple rule: **During insertion, if your current probe distance is longer than the existing key's probe distance, swap them and continue**
* Steps:

1. Each entry tracks its probe distance (distance from initial bucket/ how far from initial bucket)
2. When inserting a new entry start probing from hashed position
3. At each step compare current probe with existing entries distance from initial bucket
4. If the current distance is bigger than the existing entry, swap the entries and continue probing

* This method allows for all keys tend to have more equal probe distances and keys with long sequences get pushed forward (When inserting a new key, if it has **travelled farther** (higher probe count) than an existing key in a slot, it **takes that slot** and forces the existing key to keep probing. This "pushes" keys that were originally **close to their ideal bucket** further down the table, while keys that were **far away** move closer.)
* This is better for dynamic graphs, has more consistent performance and works well will small tables
* Cons are:

1. Slightly more overhead per operation
2. More complex implementation
3. Still needs a proper hash function and a good load factor

What our deletion entails

1. **Defunct Handling**:
   * Marks deleted slots without breaking probe sequences
   * Prevents lookup degradation from many deletions
2. **Backward Shifting**:
   * Maintains the Robin Hood property after deletion
   * Moves subsequent entries backward to fill gaps
   * Reduces average probe length over time
3. **Efficiency for Graph Operations:**
   * The water analysing will probably modify the graph frequently
   * Performance stays stable during merging/ splitting and dynamic edge updates
   * Reduces average probe length over time

* Flood-fill operations
* Even after calling iterable methods, then calling remove, the iterable will reflect this