Hashing Algorithms

Quick Review: Binary Search Trees (BST)

- Operations (all with complexity `O(h)` where `h` is the height of the tree):
  - `insert`
  - `delete`
  - `find`
  - `min/max`
  - `predecessor`
  - `successor`
- Balanced BST has a height `h = O(log n)`.

Hash Table (Generalization of arrays)

- Notation: Array `A` with `n` slots, accessed as `A[i]`.
- Hash table operations involve three steps:

1. **Pre-hashing**:
   - This step involves converting input data of various types into a uniform format that can be processed by the hash function. Typically, this means converting data into a large integer if it isn't already in numerical form.
   - Example: For string data, a common pre-hashing step is to convert each character into its ASCII value and combine these into a large integer. For instance, the string "AXY" could be converted into integers using ASCII values as follows: 'A' -> 65, 'X' -> 88, 'Y' -> 89. If we assume each character is 1 byte, the pre-hashed value might be constructed by a combination such as `65 * 256^2 + 88 * 256 + 89`.
2. **Hashing**:
   - In the hashing step, the pre-hashed value is processed by the hash function to produce an integer index in the range of possible indices for the hash table. The goal of the hash function is to distribute keys uniformly across the table to minimize collisions.
   - A common hashing technique is the division method, where the pre-hashed value is divided by the size of the hash table, and the remainder is used as the index. Other methods, like multiplication or universal hashing, are also used depending on the requirements.
3. **Post-hashing (Collision Resolution)**:

- This step deals with collisions, which occur when two different pre-hashed values produce the same hash index. Several collision resolution techniques exist:
    - **Chaining**: Each slot of the hash table contains a linked list of all elements that hash to the same index.
    - **Open Addressing**: If a collision occurs, a probing sequence is followed to find an empty slot to place the new element. Examples of probing sequences include linear probing, quadratic probing, and double hashing.

Here is a simple pseudocode that outlines these steps in the context of inserting a new element:

```
function insertIntoHashTable(table, key, value) {
    // Step 1: Pre-hash the key to convert it to a large integer
    prehashedKey = prehash(key);

    // Step 2: Apply the hash function to find the index
    index = hash(prehashedKey, table.size);

    // Step 3: Resolve any potential collision
    if (table[index] is occupied) {
        // Collision detected, apply collision resolution strategy
        index = resolveCollision(table, index, key);
    }

    // Place the value in the hash table at the resolved index
    table[index] = value;
}
```

In the above pseudocode:

- `prehash(key)` is a function that converts the key into a large integer suitable for hashing.
- `hash(prehashedKey, table.size)` is the hash function that computes the index based on the prehashed key and the size of the table.
- `resolveCollision(table, index, key)` is a function that finds the next available index using a collision resolution strategy.

Pseudocode for Hashing Classes

```
// Simple class to represent a pair of integers
class Pair {
    int x;
    int y;
}

// Hash table that maps Pair to Item
// In Java: hashCode, In Python: __hash__

// Example of a bad hash function:
// f(p) = p.x + p.y or f(p) = p.x * p.y
// This is bad because it could lead to many collisions.
// For example, Pair(2,3) and Pair(3,2) would have the same hash.

// Example of a good hash function:
// f(p) = x * 37 + y
// This reduces the chance of collisions due to the prime number factor.

// Another class example with a better hash function
class Coordinates {
    int x;
    int y;
    int z;

    // The hash function uses a prime number and powers to generate a unique number
    f(c) = x * 37^2 + y * 37 + z;
}

// Class for places with a name and position
class PlaceOfInterest {
    String name;
    Coordinates pos;

    // Hash functions for name and position
    f(name) => x * 37 + y; // Simplified example
}
```

Hash Function

- A hash function maps a large range of input to a small range `[0, m)`.

- Division method: `h(x) = x % m`.
- When increasing table size, using a prime number for `m` avoids issues.
- Load factor (utilization): `α = n / m`, where `n` is the number of items and `m` is the table size.

Multiplication Method

- A method to create a hash by multiplying the input by a constant `A` (between 0 and 1), extracting the fractional part, and then scaling it to the table size `m`.
- Visual representation of a 32-bit integer `x`:

```
|----------------- 32 bits -----------------|
|---- high order ----|---- low order ----|
```

- Extract bits from the middle of the binary representation of `x` for the hash.

Review of Hashing

- Hashing is a generalization of an array.
- **Pre-hashing**: Converts data from various types to integers.
  - Example: Converting strings or other data types to large integers.

Table Size and Load Factor

- The size of the table `m` needs to be dynamic to grow/shrink exponentially.
- Number of items in the table `n` and table size `m` determine the load factor `α = n / m`.
- A good hash function leads to unique values, whereas a bad hash function leads to collisions.

Probability in Hashing

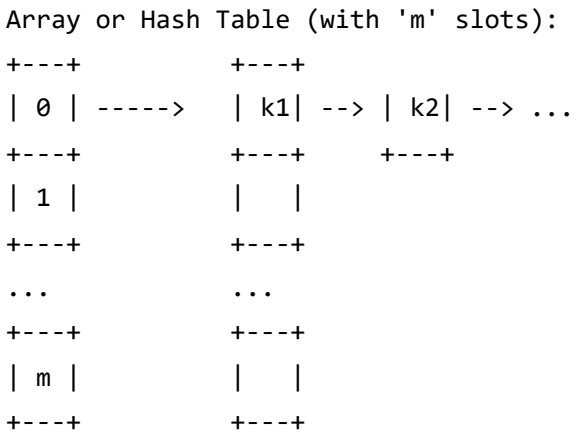
- Probability `P` that `h(k1) = h(k2)` is `1/m` when `k1 ≠ k2`.
- If `0` is not possible, then `1` is 100%.

Hash Function Efficiency

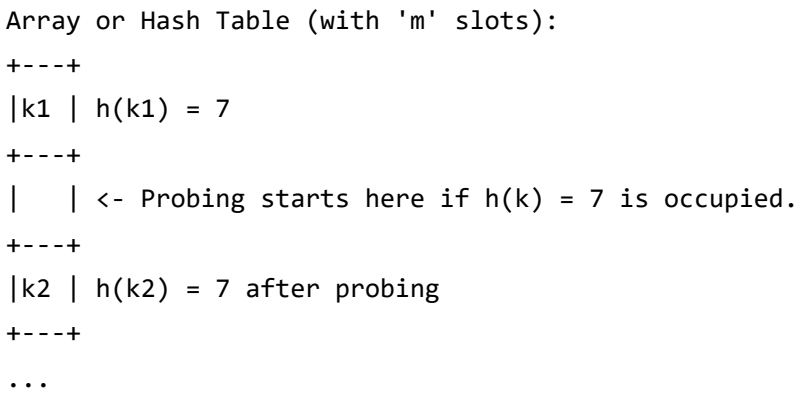
- A hash function is "perfect" when load factor `α` is very small.
- Collision will still happen; the question is how often.
- Birthday Paradox illustrates this: probability `P` that two people share the same birthday.
  - Inserting more than 23 keys can give more than a 50% chance that a collision happens.

Collision Resolution Methods

1. **Chaining**
   - Uses a linked list to handle collisions.
   - Operations `find`, `insert`, and `delete` have complexity `O(L)` where `L` is the length of the chain.
   - When the hash function `h` is "perfect", operations have `O(1)` average complexity.



   Here, 'm' is the size of the table, and each entry has a linked list.
   If h(k1) = h(k2) = 0, then both keys k1 and k2 will be in the list at slot 0.
2. **Open Addressing**
   - Linearly probes for the next empty slot.
   - Space efficiency is better than chaining.
   - Deleting is more complicated.



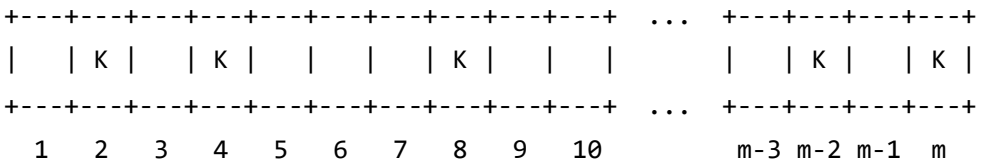
   If h(k1) is 7 and that slot is occupied, the hash table will probe linearly
   (e.g., check slot 8, then 9, etc.) until an empty slot is found for a new key.

Probability in Open Addressing

- Probability `P` for a random position being occupied: `n/m`.

- Probability `P` for a random position being empty: `1 - n/m`.
- For example, if a hash table has 'm' slots and 'n' keys have been inserted, then the probability that a new key will hash to an occupied slot (and thus a collision will occur) is `n/m`.
- The probability `P` for a random position being empty is simply the complement of the table being occupied, which is `1 - n/m`.

Visualization of Probability



Key 'K' is placed randomly; empty slots are represented by a blank space.

- The coin toss analogy illustrates the likelihood of collision in simple terms. If you flip a coin, the chance of getting heads (`H`) might be `1/100`, and tails (`T`) `99/100`. Relating this to hashing, the chance of a collision could be seen as the chance of getting heads.