Repl.it

- https://replit.com/@rugbyprof/weak-cryptographic-hash-example
- https://replit.com/@rugbyprof/HelloWorldHashTable
- https://replit.com/@rugbyprof/polynomialhash
- https://replit.com/@rugbyprof/WorstHashFunction2019

Primes

- http://compoasso.free.fr/primelistweb/page/prime/liste_online_en.php

Hash Functions

- The hash functions are typically designed to return a value in the full unsigned range of an integer.
- For a 32-bit integer, this means that the hash functions will return a value in the range `[0..4,294,967,296)`

Avalanche

The term "avalanche effect" in the context of hash functions is crucial because it describes how a small change to the input (such as changing a single bit) should result in a significant and unpredictable change in the output hash value. This characteristic is important for several reasons:

1. **Uniform Distribution**: The avalanche effect helps ensure that hash values are distributed uniformly across the hash table. This uniform distribution minimizes collisions (different inputs hashing to the same output), which is key for the efficiency of data retrieval and insertion in hash-based structures like hash tables.

2. **Security**: In cryptographic hash functions, the avalanche effect is critical for security. It makes it difficult to predict the hash for a similar input or to find two inputs that produce the same hash (collision resistance). This unpredictability is vital for cryptographic applications, including data integrity checks, digital signatures, and password storage.

3. **Minimizing Clustering**: By ensuring that small changes can significantly alter the output, the avalanche effect helps prevent clustering of hash values, where many inputs would hash to a few output values, leading to performance degradation in hash tables.

The avalanche effect is a desirable property of hash functions that enhances their performance and security by ensuring that outputs are well-distributed and unpredictable based on the input.

Simple hash functions

The following functions map a single integer key `(k)` to a small integer bucket value `h(k)`. `m` is the size of the hash table (number of buckets).

Division method (Cormen)

The Multiplication Method for hashing is a technique presented by Thomas H. Cormen and his co-authors in the widely recognized textbook, "Introduction to Algorithms". This method is designed to distribute a set of keys uniformly into hash table slots using a simple multiplication operation. Here's a brief overview of how it works:

Basic Idea

The Multiplication Method computes a hash value by taking the key, multiplying it by a constant fraction, and then extracting a specific part of the result as the hash value. The method is favored for its simplicity and the fact that it does not require knowledge of the hash table size being a power of two, making it versatile and easy to implement.

Process

Given a key (k), the hash function (h(k)) is computed as follows:

1. **Multiply** the key (k) by a constant (A) (0 < (A) < 1), which is not necessarily related to the size of the hash table.
2. **Extract** the fractional part of (kA).
3. **Multiply** this value by the hash table size (m).
4. **Take the floor** of the result to get the final hash value.

Mathematically, this can be expressed as:

[h(k) = \lfloor m(kA \mod 1) \rfloor]

where (\mod 1) means to take the fractional part of (kA), and (\lfloor \cdot \rfloor) denotes the floor operation, which rounds the number down to the nearest integer.

Choice of (A)

The choice of the constant (A) significantly affects the hash function's ability to distribute keys evenly. Cormen et al. suggest that the golden ratio, either (\phi) or its conjugate, can serve as an effective choice for (A), due to its desirable mathematical properties. Specifically, (A) can be set to (\frac{\sqrt{5} - 1}{2} \approx 0.6180339887...), which is the fractional part of the golden ratio.

Advantages

- **Simplicity**: The Multiplication Method is straightforward to implement.
- **Good Distribution**: It tends to produce a uniform distribution of hash values for a typical set of input keys.
- **Independence from Table Size**: It does not require the hash table size to be a prime number or a power of two, offering flexibility in table size selection.

Application

The Multiplication Method is well-suited for situations where the distribution of input keys is unknown or when there's a need for a simple yet effective hashing mechanism. It's particularly useful in the design of hash tables where minimizing collisions and achieving a uniform distribution of keys across the hash table are desired objectives.

This hashing method's effectiveness and its ability to be easily adapted to different scenarios make it a popular choice in various applications, from database indexing to in-memory data structures.

Knuth Variant on Division

- `h(k) = k(k+3) mod m`.
- Supposedly works much better than the raw division method.

Multiplication Method (Cormen)

The Division Method for hashing is another approach described by Thomas H. Cormen and his co-authors in the textbook "Introduction to Algorithms." This method is a straightforward technique to map a universe of keys into the slots of a hash table. Here's a concise overview of how the Division Method works:

Basic Concept

The Division Method generates a hash index by dividing the key (k) by the size of the hash table (m) and then taking the remainder as the hash value. The essence of this method lies in its simplicity and the direct use of modular arithmetic to ensure that the hash values are evenly distributed across the hash table slots, given a suitably chosen table size.

Process

Given a key (k), the hash function (h(k)) under the Division Method is calculated as:

[h(k) = k \mod m]

where

- (k) is the key to be hashed,
- (m) is the size of the hash table,
- and (\mod) denotes the modulo operation, which yields the remainder of the division of (k) by (m).

Choice of (m)

The effectiveness of the Division Method largely depends on the choice of (m), the hash table size. To achieve a uniform distribution of hash values:

- (m) should not be a power of 2, because if (m) is (2^n), then (h(k)) would simply be the (n) least significant bits of (k), which might not be uniformly distributed for typical data.
- (m) should ideally be a prime number that is not too close to a power of 2. Choosing (m) as a prime number helps in reducing the likelihood of collisions (i.e., different keys hashing to the same value) and thus leads to a more uniform distribution of keys across the hash table.

Advantages

- **Ease of Implementation**: The Division Method is straightforward and easy to implement, making it an attractive first option for hash function design.
- **Uniform Distribution**: With a well-chosen table size (m), this method can provide a good distribution of hash values, especially when the keys are random.

- **Efficiency**: The computation of the hash value is efficient, requiring only a single division operation.

Application

The Division Method is particularly useful for constructing hash functions when the distribution of the input keys is uniform or when there is little knowledge about the characteristics of the input keys. It's widely applied in scenarios ranging from in-memory data structures, like hash tables and hash maps, to database indexing, where efficient and straightforward hashing is required.

Conclusion

While the Division Method is celebrated for its simplicity, the choice of the hash table size (m) is crucial for its success in minimizing collisions and achieving an even distribution of keys. This method provides a solid foundation for hash function design, especially in applications where simplicity and efficiency are prioritized.

Table Size

Now let's illustrate the importance of uniform distribution in hash tables and the impact of table size selection with a simple example. We'll use a basic division-based hash function (h(k) = k \mod m), where (k) is the key and (m) is the table size.

Scenario: Poorly Chosen Table Size

Let's say we have a hash table with a size (m = 10) (a poor choice because it's a round number, making it more prone to patterns in the data affecting distribution). Our keys will be simple integers, and we'll see how they distribute with such a table size.

Example Keys

Consider we have a set of keys that are multiples of 5: (5, 10, 15, 20, 25, 30, ...). This is not an unusual scenario, as keys could represent calculated or measured values that happen to have a common factor, especially in specific domains or due to data characteristics.

Hashing the Keys

Using our hash function (h(k) = k \mod 10), let's see how these keys are distributed:

- (h(5) = 5 \mod 10 = 5)
- (h(10) = 10 \mod 10 = 0)
- (h(15) = 15 \mod 10 = 5)
- (h(20) = 20 \mod 10 = 0)
- (h(25) = 25 \mod 10 = 5)
- (h(30) = 30 \mod 10 = 0)

Observations

Notice that all keys are hashed to only two locations in the hash table: slots 0 and 5. This results in a severe underutilization of the table's capacity and leads to clustering, where multiple keys hash to the same slots, increasing the likelihood of collisions.

Visual Representation

Imagine a hash table visualized as ten buckets in a row, labeled from 0 to 9. In our example, only the buckets labeled 0 and 5 would have keys in them, while the others remain empty. This visualization helps convey the inefficiency and potential performance problems with such a hash table configuration.

Discussion

This example demonstrates that choosing a table size (m) that is a simple round number (especially one that might commonly be a factor of keys) can lead to poor distribution of keys. It highlights the significance of selecting a prime number for (m), ideally one that does not closely follow the form (2^n) or (2^n - 1), to help ensure a more uniform distribution of keys across all available slots in the hash table, regardless of any patterns in the data.

Conclusion

The key takeaway should be the choice of a hash table size and hash function directly impacts the efficiency of data retrieval and storage in the hash table. By choosing a prime number for the table size and designing hash functions that distribute keys uniformly, we can minimize collisions and maximize the performance of hash-based data structures. This lesson is critical for understanding how to implement effective hash tables that perform well under various data conditions.

Polynomomial Hash

A polynomial hash function is a method of generating a hash value from a string (or generally, a sequence of characters) by treating each character as a coefficient in a polynomial of some fixed degree, evaluated at a certain point. This approach is widely used in string hashing and has applications in string matching algorithms, data storage, and retrieval systems.

Basic Concept

The idea behind a polynomial hash function is to map a string (s) of length (n), composed of characters (s_1, s_2, ..., s_n), to a numerical value. The hash function typically looks like this:

[h(s) = s_1a^{n-1} + s_2a^{n-2} + ... + s_{n-1}a + s_n \mod m]

where:

- (h(s)) is the hash value of string (s).
- (s_i) represents the numerical value of the (i)th character in the string.
- (a) is a constant, and (a > 1). Often, (a) is chosen to be a prime number to ensure a good distribution of hash values.
- (m) is a large prime number, used as the modulus to ensure the hash value fits into a fixed size and to reduce collisions. Taking the result modulo (m) also helps in keeping the hash value within a specific range, such as the size of a hash table.
- The exponent (n-i) ensures that the position of each character influences the hash value.

Properties and Usage

1. **Efficiency**: Polynomial hash functions are efficient to compute, especially when using Horner's method to evaluate the polynomial. This method reduces the computational complexity by turning the polynomial into a nested form, allowing the hash to be computed iteratively in linear time relative to the string length.

2. **Uniform Distribution**: If the constants (a) and (m) are chosen carefully, polynomial hash functions can achieve a uniform distribution of hash values over the output space, reducing the likelihood of collisions.

3. **Applicability**: Polynomial hashing is particularly useful in algorithms that require comparing substrings within a larger string, such as in the Rabin-Karp string search algorithm. The choice of the base (a) and modulus (m) is critical in minimizing collisions and ensuring efficient and reliable hashing.

Example

Consider a simple string "abc" with characters encoded in ASCII ('a' = 97, 'b' = 98, 'c' = 99), a base (a = 101), and a large modulus (m). The hash value (h("abc")) can be calculated as:

[h("abc") = (97 \times 101^2 + 98 \times 101^1 + 99 \times 101^0) \mod m]

This method turns the string into a large numerical value based on the polynomial calculation, then takes it modulo (m) to produce the final hash value.

Security Note

While polynomial hash functions are widely used for non-cryptographic applications due to their efficiency and simplicity, they are generally not suitable for cryptographic purposes without additional security considerations, due to vulnerability to collision attacks if the adversary can control the input.

Hashing sequences of characters

The hash functions in this section take a sequence of integers $k=k_1,...,k_n$ and produce a small integer bucket value **h(k)**. **m** is the size of the hash table (number of buckets), which should be a prime number. The sequence of integers might be a list of integers or it might be an array of characters (a string).

The specific tuning of the following algorithms assumes that the integers are all, in fact, character codes.

- In C++, a character is a char variable which is an 8-bit integer.
- ASCII uses only 7 of these 8 bits. Of those 7, the common characters (alphabetic and number) use only the low-order 6 bits. And the first of those 6 bits primarily indicates the case of characters, which is relatively insignificant.
- So the following algorithms concentrate on preserving as much information as possible from the last 5 bits of each number, and make less use of the first 3 bits.

When using the following algorithms, the inputs k_i must be unsigned integers. Feeding them signed integers may result in odd behavior.

For each of these algorithms, let h be the output value. Set h to 0. Walk down the sequence of integers, adding the integers one by one to h. The algorithms differ in exactly how to combine an integer ki with h. The final return value is h mod m.

CRC variant:

Do a 5-bit left circular shift of h. Then XOR in k_i. Specifically:

```
    highorder = h & 0xf8000000    // extract high-order 5 bits from h
                                  // 0xf8000000 is the hexadecimal
representation
                                  //   for the 32-bit number with the
first five
                                  //   bits = 1 and the other bits = 0
    h = h << 5                    // shift h left by 5 bits
    h = h ^ (highorder >> 27)     // move the highorder 5 bits to the
low-order
                                  //   end and XOR into h
    h = h ^ ki                    // XOR h and ki
```

PJW hash:

Left shift h by 4 bits. Add in k_i. Move the top 4 bits of h to the bottom. Specifically:

```
    // The top 4 bits of h are all zero
    h = (h << 4) + ki               // shift h 4 bits left, add in ki
    g = h & 0xf0000000              // get the top 4 bits of h
    if (g != 0)                     // if the top 4 bits aren't zero,
       h = h ^ (g >> 24)            //   move them to the low end of h
       h = h ^ g
    // The top 4 bits of h are again all zero
```

PJW and the CRC variant both work well and there's not much difference between them. We believe that the CRC variant is probably slightly better because

- It uses all 32 bits. PJW uses only 24 bits. This is probably not a major issue since the final value m will be much smaller than either.
- 5 bits is probably a better shift value than 4. Shifts of 3, 4, and 5 bits are all supposed to work OK.
- Combining values with XOR is probably slightly better than adding them. However, again, the difference is slight.

BUZ hash:

Set up a function ***R*** that takes 8-bit character values and returns random numbers. This function can be precomputed and stored in an array. Then, to add each character k_i to ***h***, do a 1-bit left circular shift of ***h*** and then `XOR` in the random value for k_i. That is:

```
    highorder = h & 0x80000000    // extract high-order bit from h
    h = h << 1                    // shift h left by 1 bit
    h = h ^ (highorder >> 31)     // move them to the low-order end and
                                  // XOR into h
    h = h ^ R[ki]                 // XOR h and the random value for ki
```

Rumor has it that you may have to run a second hash function on the output to make it random enough. Experimentally, this function produces good results, but is a bit slower than the CRC variant and PJW.

Additive Hash

```
//-------------------------------------------------------------------
// Additive Hash
// Adds all of the characters together using ascii values.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//------------------------------------------------------------------
unsigned int hash_functions::add_hash (string val)
{
  unsigned int h = 0;

  for (unsigned int  i = 0; i < val.length(); i++ )
    h += val[i];

  return h;
}
```

XOR hash

```
//-------------------------------------------------------------------
// XOR hash
// Repeatedly folds the bytes together using the XOR operation to
// produce a seemingly random hash value.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//------------------------------------------------------------------
unsigned int hash_functions::xor_hash (string val)
{
  unsigned h = 0;

  for (unsigned i = 0; i < val.length(); i++ )
    h ^= val[i];

  return h;
}
```

Rotating hash

```
//-----------------------------------------------------------------
// Rotating hash
// The rotating hash is identical to the XOR hash except instead of simply
// folding each byte of the input into the internal state, it also
performs
// a fold of the internal state before combining it with the each byte of
// the input
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//----------------------------------------------------------------
unsigned int hash_functions::rot_hash (string val)
{
  unsigned h = 0;

   for (unsigned i = 0; i < val.length(); i++ )
      h = ( h << 4 ) ^ ( h >> 28 ) ^ val[i];

  return h;
}
```

Bernstein hash

```
//-----------------------------------------------------------------
// Bernstein hash
// Dan Bernstein created this algorithm and posted it in a newsgroup.
// It is known by many as the Chris Torek hash because Chris went a
// long way toward popularizing it. Since then it has been used
// successfully by many, but despite that the algorithm itself is not
// very sound when it comes to avalanche and permutation of the internal
// state. It has proven very good for small character keys, where it
// can outperform algorithms that result in a more random distribution.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//----------------------------------------------------------------
unsigned int hash_functions::bernstein_hash (string val)
{
  unsigned h = 0;

    for (unsigned i = 0; i < val.length(); i++ )
      h = 33 * h + val[i];

  return h;
}
```

Modified Bernstein hash

```
//-----------------------------------------------------------------
// Modified Bernstein hash
// A minor update to Bernstein's hash replaces addition with XOR for
// the combining step. This change does not appear to be well known
// or often used, the original algorithm is still recommended by
// nearly everyone, but the new algorithm typically results in a
// better distribution.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//----------------------------------------------------------------
unsigned int hash_functions::mod_bernstein_hash (string val)
{
  unsigned h = 0;

    for (unsigned i = 0; i < val.length(); i++ )
      h = 33 * h ^ val[i];

  return h;
}
```

Shift-Add-XOR hash

```
//-----------------------------------------------------------------
// Shift-Add-XOR hash
// The shift-add-XOR hash was designed as a string hashing function,
// but because it is so effective, it works for any data as well with
// similar efficiency. The algorithm is surprisingly similar to the
// rotating hash except a different choice of constants for the rotation
// is used, and addition is a preferred operation for mixing. All in
// all, this is a surprisingly powerful and flexible hash. Like many
// effective hashes, it will fail tests for avalanche, but that does
// not seem to affect its performance in practice.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//----------------------------------------------------------------
unsigned int hash_functions::shift_add_xor_hash (string val)
{
  unsigned h = 0;

   for (unsigned i = 0; i < val.length(); i++ )
      h ^= ( h << 5 ) + ( h >> 2 ) + val[i];

  return h;
}
```

FNV hash

```
//-----------------------------------------------------------------
// FNV hash
// The FNV hash, short for Fowler/Noll/Vo in honor of the creators,
// is a very powerful algorithm that, not surprisingly, follows the
// same lines as Bernstein's modified hash with carefully chosen
// constants. This algorithm has been used in many applications with
// wonderful results, and for its simplicity, the FNV hash should be
// one of the first hashes tried in an application. It is also
// recommended that the FNV website be visited for useful descriptions
// of how to modify the algorithm for various uses.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//----------------------------------------------------------------
unsigned int hash_functions::fnv_hash (string val)
{
  unsigned int h = 2166136261;

   for (unsigned i = 0; i < val.length(); i++ )
      h = ( h * 16777619 ) ^ val[i];

  return h;
}
```

One-at-a-Time hash

```
//-----------------------------------------------------------------
// One-at-a-Time hash
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//----------------------------------------------------------------
unsigned int hash_functions::one_at_a_time_hash (string val)
{
  unsigned int h = 0;

  for (unsigned i = 0; i < val.length(); i++ ) {
    h += val[i];
    h += ( h << 10 );
    h ^= ( h >> 6 );
  }

  h += ( h << 3 );
  h ^= ( h >> 11 );
  h += ( h << 15 );

  return h;
}
```

Jsw hash

```
//-----------------------------------------------------------------
// Jsw hash
// This is a hash of my own devising that combines a rotating hash
// with a table of randomly generated numbers. The algorithm walks
// through each byte of the input, and uses it as an index into a
// table of random integers generated by a good random number
// generator. The internal state is rotated to mix it up a bit, then
// XORed with the random number from the table. The result is a
// uniform distribution if the random numbers are uniform. The size
// of the table should match the values in a byte. For example, if a
// byte is eight bits then the table would hold 256 random numbers.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//----------------------------------------------------------------
unsigned int hash_functions::jsw_hash (string val)
{
  unsigned h = 16777551;

  for (unsigned i = 0; i < val.length(); i++ )
    h = ( h << 1 | h >> 31 ) ^ rand_vals[val[i]];

  return h;
}
```

Elf Hash

```


//-----------------------------------------------------------------
// Elf hash
// The ELF hash function has been around for a while, and it is believed
// to be one of the better algorithms out there. Though ELF hash does not
// perform sufficiently better than most of the other algorithms
// in this collection to justify its slightly more complicated
implementation.
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//----------------------------------------------------------------
unsigned int hash_functions::elf_hash (string val)
{
  unsigned h = 0 , g;

    for (unsigned i = 0; i < val.length(); i++ ) {
      h = ( h << 4 ) + val[i];
      g = h & 0xf0000000L;

      if ( g != 0 )
        h ^= g >> 24;

      h &= ~g;
    }
    return h;
}
```

The ELF hash function is a good example of a hash algorithm that combines simple arithmetic and bitwise operations to distribute input (e.g., strings) across a wide range of hash values. Here's a step-by-step breakdown of how the ELF hash function works, using the implementation you've provided:

Step-by-Step Algorithm of ELF Hash Function

1. **Initialization**:
    - Start with a hash value `h` set to 0. This will accumulate the result.
2. **Iteration Over Each Character**:
    - For each character `c` in the input string `val`, repeat the following steps: a. **Update Hash**: Multiply `h` by 16 (equivalent to left shifting by 4 bits, `h << 4`) and then add the ASCII value of the current character `c` to `h`. This step gradually builds up the hash value based on the entire string's characters. b. **Calculate High Order Bits**: Determine the high-order bits of `h` (those beyond the 28th bit) by performing a bitwise AND operation between `h` and `0xf0000000L`. This extracts the top four bits of `h`, stored in `g`. c. **Conditional Mixing**:
        - If `g` is not zero (meaning there were significant high-order bits in `h`), mix these bits back into the hash by XORing `h` with `g` shifted right 24 places (`g >> 24`). This step disperses the influence of the high-order bits back into the rest of the hash value to ensure even distribution. d. **Clear High Order Bits**: Clear the high-order bits in `h` that were stored in `g`. This is achieved by ANDing `h` with the negation of `g` (`h &= ~g`), effectively setting the top four bits to zero and preventing overflow in subsequent iterations.
3. **Return Hash**: After processing all characters in the string, return the accumulated hash value `h`.

Explanation of Key Operations

- **Bit Shifting (`<<` and `>>`)**: Bit shifting is used to efficiently multiply and divide by powers of two. Left shifting a value by `n` (`val << n`) multiplies it by (2^n), while right shifting (`val >> n`) divides it by (2^n), using integer division.
- **Bitwise AND (`&`)**: The AND operation is used to mask certain bits, i.e., to extract or clear specific bits within a value. In this case, it's used to extract the high-order bits (`h & 0xf0000000L`) and to clear them later (`h &= ~g`).
- **Bitwise XOR (`^`)**: XOR is a bitwise operation that mixes bits. If `g` is not zero, XORing `h` with the right-shifted `g` disperses these high-order bits back into `h`, helping to improve the distribution of hash values.
- **Negation (`~`)**: The negation operator flips all the bits of a value. It's used here to create a mask that clears the high-order bits in `h`.

Visualization

Imagine the hash value `h` as a container being filled with contributions from each character in the string. The process ensures that both the character's value and its position influence the final hash, while also carefully managing the high-order bits to prevent overflows and to ensure that early characters influence the final result.

This step-by-step breakdown abstracts away the specific bit-level operations to focus on the algorithm's logic and goals: accumulating influence from each character, managing overflow potential, and ensuring a uniform distribution of hash values.

My Hash

```
//------------------------------------------------------------------
// My hash
// I basically took ideas from above hash functions and others found
// on the internet to create my own. this function uses the rotative
// and additive priniciple to create a unique hash key for every input
// @Param: string val - word to be hashed
// @Returns: unsigned int - hash key value
//-----------------------------------------------------------------
unsigned int hash_functions::my_hash (string val)
{
     unsigned int hash = 0xAAAAAAAA;

     for(unsigned i = 0; i < val.length(); i++)
    {
       hash ^= ((i & 1) == 0) ? (  (hash <<  11) ^ val[i] * (hash >> 7)) :
                               (~((hash << 19) + (val[i] ^ (hash >>
11))));
    }

    return hash;
}
```

🎯 Example Hash Functions - A Lecture on Hashing in C++

📝 Introduction

A **hash function** is a mathematical function that **converts input data (keys) into an index** in a **hash table**. The goal of a good hash function is to distribute keys **evenly** and **efficiently** while minimizing **collisions**.

- This lecture will cover:
  - What makes a **good hash function**.
  - **Common hash function techniques**.
  - **Example implementations** in C++.

🚀 1. What Makes a Good Hash Function?

- A **good hash function** should:

1. **Be deterministic** → The same input always produces the same output.
2. **Be fast** → The function should execute in **constant time O(1)**.
3. **Distribute keys uniformly** → Prevent clustering in a few spots.
4. **Minimize collisions** → Different inputs should hash to different values as much as possible.
5. **Use the entire range of the table** → Spread values evenly.

🔢 2. Example Hash Functions in C++

✨ 1. Simple Modulo Hashing

A basic way to map integer keys to a hash table of size N is using **modulo division**.

Formula:

$\text{index} = (\text{key} \mod \text{table size})$

C++ Implementation:

```cpp
#include <iostream>
using namespace std;

int hashFunction(int key, int tableSize) {
    return key % tableSize;  // Modulo operation to get an index
}

int main() {
    int tableSize = 10;
    cout << "Hash of 25: " << hashFunction(25, tableSize) << endl;
    cout << "Hash of 37: " << hashFunction(37, tableSize) << endl;
    cout << "Hash of 49: " << hashFunction(49, tableSize) << endl;
    return 0;
}
```

🛠️ Output Example:

```
Hash of 25: 5
Hash of 37: 7
Hash of 49: 9
```

- ✅ Pros: Simple, fast, and efficient for integer keys.
- ❌ Cons: Can cause collisions when numbers cluster around the same remainders.

✨ 2. Multiplication Method

This method is useful when the key space is large.

Formula:

$\text{index} = \lfloor N \times (K \times A \mod 1) \rfloor$

Where: • K is the key. • A is a constant fractional number (commonly 0.6180339887). • N is the table size. • mod 1 extracts the decimal portion.

C++ Implementation:

#include using namespace std;

int hashFunction(int key, int tableSize) { const double A = 0.6180339887; // A fractional constant (commonly used) double fractionalPart = key * A - int(key * A); return int(tableSize * fractionalPart); }

int main() { int tableSize = 10; cout << "Hash of 25: " << hashFunction(25, tableSize) << endl; cout << "Hash of 37: " << hashFunction(37, tableSize) << endl; cout << "Hash of 49: " << hashFunction(49, tableSize) << endl; return 0; }

✅ Pros: Works well for non-uniform data, spreads keys better than modulo. ❌ Cons: Slightly more computation than modulo.

✨ 3. String Hashing (Polynomial Rolling Hash)

When hashing strings, each character contributes to the final value.

Formula:

[\text{hash} = (c_1 \times p^0 + c_2 \times p^1 + c_3 \times p^2 + \dots) \mod M] Where: • c_i = ASCII value of character at position i. • p = Small prime number (e.g., 31). • M = Large prime modulus to prevent overflow (e.g., 1e9 + 9).

C++ Implementation:

#include using namespace std;

const int P = 31; const int MOD = 1e9 + 9;

int stringHash(string s, int tableSize) { long long hashValue = 0; long long pPower = 1;

```
for (char c : s) {
    hashValue = (hashValue + (c - 'a' + 1) * pPower) % MOD;
    pPower = (pPower * P) % MOD;
}
return hashValue % tableSize;
```

}

int main() { int tableSize = 10; cout << "Hash of 'sword': " << stringHash("sword", tableSize) << endl; cout << "Hash of 'shield': " << stringHash("shield", tableSize) << endl; cout << "Hash of 'staff': " << stringHash("staff", tableSize) << endl; return 0; }

✅ Pros: Works well for variable-length strings, commonly used in text search & hashing algorithms. ❌ Cons: Computationally heavier than integer hashing.

✨ 4. Universal Hashing (Randomized)

Universal hashing chooses a random hash function at runtime from a set of hash functions.

Formula:

[\text{hash} = ((a \times key + b) \mod p) \mod N] Where: • a, b are randomly chosen constants. • p is a prime number greater than N. • N is the table size.

C++ Implementation:

#include #include using namespace std;

int universalHash(int key, int tableSize) { const int p = 101; // Large prime number int a = rand() % p + 1; // Random number in range [1, p-1] int b = rand() % p; // Random number in range [0, p-1] return ((a * key + b) % p) % tableSize; }

int main() { int tableSize = 10; cout << "Hash of 25: " << universalHash(25, tableSize) << endl; cout << "Hash of 37: " << universalHash(37, tableSize) << endl; cout << "Hash of 49: " << universalHash(49, tableSize) << endl; return 0; }

✅ Pros: Prevents worst-case attacks, great for cryptographic applications. ❌ Cons: Requires randomization, slightly slower due to modular arithmetic.

🎯 5. Comparing Hash Functions

Hashing Method Pros Cons Best Used For Modulo Hashing Fast, simple Prone to clustering Small integers Multiplication Hashing More uniform spread Requires floating-point ops Large key spaces Polynomial Hashing Works well for strings Requires prime modulus Text search, dictionaries Universal Hashing Prevents worst-case attacks Needs randomness Cryptographic applications

🎯 6. Summary & Key Takeaways • A good hash function should be fast, uniform, and minimize collisions. • Modulo Hashing is simple but may cause clustering. • Multiplication Hashing helps with uniform distribution. • String Hashing (Polynomial Rolling) is great for text-based data. • Universal Hashing is best for cryptographic and security applications.

📝 Homework & Exercises 1. Modify the modulo hashing function to work with floating-point numbers. 2. Implement a simple hash table using chaining with modulo hashing. 3. Test different string hashing techniques using real-world data.

🚀 Happy Hashing!