These are six different hash functions that I will be using and analyzing in this assignment.

1. Polynomial Hash Function
2. Summation Hash Function
3. Cycle Shift Hash Function
4. Polynomial & Cycle Shift Hash Function
5. Summation & Cycle Shift Hash Function
6. Summation & Polynomial Hash Function

1. Polynomial Hash Function

Pseudocode

```
Algorithms: polynomialHash(key)
  Input: string key

  unsigned int myHashCode ← 0
  unsigned int a ← 37
  unsigned int b ← 1

 for each character ch in key do
    myHashCode ← myHashCode + (unsigned int) *iterator * b
    b ← a * b

 unsigned int x ← 31
 unsigned int y ← 7

 unsigned int finalHashCode ← (x * myHashCode + y) modulo capacity

 return finalHashCode
```

Implementation in C++

```cpp
//Function with polynomial hashing techniques
unsigned int HashTable::polynomialHash(const string& key){
   //Initialize myHashCode to 0
   unsigned int myHashCode = 0;
   //Initialize a to 37
   unsigned int a = 37;
   //Initialize b to 1
   unsigned int b = 1;

   //Iterate through each character in the given word
   for (std::string::const_iterator iterator = key.begin(); iterator != key.end(); ++iterator){
      //Cast each char into int and multiply hash code by a
      myHashCode += ((unsigned int) *iterator) * b;
      //Iteratively raise the power of a
      b = a * b;
   }

   //Initialize x to 31
   const unsigned int x = 31;
   //Initialize y to 7
   const unsigned int y = 7;
```

```
    //Get the hash code by making it fall into the range between 0 and capacity - 1
    unsigned int finalHashCode = ((x * myHashCode + y) % capacity);

    //Return hash code
    return finalHashCode;
}
```

2. Summation Hashing Function

Pseudocode

Algorithm: SummationHash(key)
Input: string key
  unsigned int myHashCode ← 0
  for each character ch in key do
    hash ← hash + (unsigned int) *iterator

  unsigned int hashCode ← myHashCode modulo capacity

  return hashCode

Implementation in C++

```
//Function using Summation hashing techniques
unsigned int HashTable::SummationHash(const std::string& key){
  //Initialize hash code to 0
  unsigned int myHashCode = 0;

  //Iterate through each character in the given word and take the sum after casting each of them into integer
  for (std::string::const_iterator iterator = key.begin(); iterator != key.end(); ++iterator){
    //Cast char into int and sum them up (ASCII value)
    myHashCode += (unsigned int) *iterator;
  }

  //Take the remainder by using modulo to make hash code fall into the range between 0 and capacity - 1
  unsigned int hashCode = myHashCode % capacity;

  //Return hash code
  return hashCode;
}
```

3. Cycle Shift Hash Function

Pseudocode

Algorithm: CycleShiftHash (key)
Input: string key

```
unsigned int h ← 0;

for each character ch in key do
    char ch ← *iterator
    h ← (h shifted left by 5 bits) OR  (h shifted right by 27 bits)
    h ← h XOR ch

unsigned int hashCode ← h modulo capacity

return hashCode
```

Implementation in C++

```cpp
//Cycle shift function to generate the hash code using bitwise operations
unsigned int HashTable::CycleShiftHash(const std::string& key){
    //Initialize hash code to 0
    unsigned int h = 0;

    //Iterate through each character in the string using iterator
    for (std::string::const_iterator iterator = key.begin(); iterator != key.end(); ++iterator){
        //Dereference iteratorn to get the character
        char ch = *iterator;
        //Shift hash code 5 bits to the left and 27 bits to the right
        h = (h << 5) | (h >> 27);
        //Combine the results using bitwise or operator
        h ^= ch;
    }

    //Use Division method for compress function
    unsigned int hashCode = h % capacity;

    //Return the hash code
    return hashCode;
}
```

4. Polynomial & Cycle Shift Hashing Function

This function is a combination of polynomial hashing and cycle shift hashing, aimed at leveraging the strengths of both techniques to improve the efficiency of handling collisions. The MAD method at the end is also used to further enhance the efficiency.

The way this hash function works is that it first carries out the polynomial hashing technique to generate an ASCII value of each word by incorporating a constant multiplier based on the position of each character. Subsequently, the hash value (polyHash) goes through a cycle shift hashing phase, where each bit is scrambled by being shifted 5 bits to the left and 27 bits to the right, followed by the XOR operation to increase randomness. Finally, the compress function using the MAD method will fit the hash values into the range of the hash table's capacity.

Pseudocode

```
Algorithm: PolyCyclicHash (key)
Input: string key

  unsigned int myHashCode ← 0
  unsigned long myHashCode ← 0
  unsigned long a ← 39
  unsigned long b ← 1

  for each character ch in key do
    myHashCode ← myHashCode + ((unsigned int) *iterator) * b
    b ← (b * a)

  unsigned int finalHash ← 0
  unsigned int polyHash ← myHashCode

  for each bit i in polyHash do
    unsigned int bit ← (polyHash shifted right by i bits) AND 1
    finalHash ← (finalHash shifted left by 5 bits) OR (finalHash shifted right by 27 bits)
    finalHash ← finalHash XOR bit

 const unsigned int x ← 31
 const unsigned int  y ← 3

 unsigned int hashCode ← (finalHash * x + y) modulo capacity

 return hashCode
```

Implementation in C++

```
//Function with both polynomial and cycle shift hashing techniques
unsigned int HashTable::PolyCyclicHash(const std::string& key){
   //Initialize hash code to 0
   unsigned int myHashCode = 0;
   //Initialize a to 39
   unsigned int a = 39;
   //Initialize b to 1
   unsigned int b = 1;

   //Iterate through each character in the given word
   for (std::string::const_iterator iterator = key.begin(); iterator != key.end(); ++iterator){
      //Sum up the result of hash codes multiplied by b
      myHashCode += ((unsigned int) *iterator) * b;
      //Iteratively raise the power of a
      b = a * b;
   }

   //Initialize hash code to 0
   unsigned int finalHash = 0;
   //Store the result from polynomial hashing phase
   unsigned int polyHash = myHashCode;

   //Iterate through each bit in the polyHash
   for (unsigned int i = 0; i < sizeof(polyHash) * 8; i++){
      //Store each bit
      unsigned int bit = (polyHash >> i) & 1;
      //Shift hash code 5 bits to the left and 27 bits to the right
      finalHash = (finalHash << 5) | (finalHash >> 27);
      //Combine the results using XOR operator
      finalHash ^= bit;
   }

   //Initialize x to 31
   const unsigned int x = 31;
   //Initialize y to 3
   const unsigned int y = 3;
   //Take the remainder by using modulo to make hash code fall into the range between 0 and capacity - 1
   unsigned int hashCode = ((finalHash * x + y) % capacity);

   //Return hash code
   return hashCode;
}
```

5. Summation Hashing & Cycle Shift Hashing Function

This hash function is a combination of summation hashing and cycle shift hashing, aimed at increasing the efficiency of handling collisions compared to the summation hash function.

The way this function works is that it first carries out the summation hashing technique where it produces the sum of the ASCII values of each word. Then, the hash code will go through a cycle shift hashing, where each bit of the hash code is scrambled by being shifted 5 bits to the left and 27 bits to the right, followed by the XOR operation to increase the randomness. Finally, the compress function using division method aims to further enhance efficiency.

Pseudocode

```
Algorithm: SumCyclicHash (key)
Input: string key

  unsigned int myHashCode ← 0

  for each character ch in key do
     myHashCode ← myHashCode + (unsigned int) *iterator

  unsigned int finalHash ← 0
  unsigned int sumResult ← myHashCode

  for each bit i in sumResult do
    unsigned int bit ← (sumResult shifted right by i bits) AND 1
    finalHash ← (finalHash shifted left by 5 bits) OR (finalHash shifted right by 27 bits)
    finalHash ← finalHash XOR bit

  unsigned int hashCode ← finalHash modulo capacity

  return hashCode
```

Implementation in C++

```
//Hash Function with summation and cycle shift hashing techniques
unsigned int HashTable::SumCyclicHash(const std::string& key){
   //Initialize hash code to 0
   unsigned int myHashCode = 0;
   //Iterate through each character in the given word
   for (std::string::const_iterator iterator = key.begin(); iterator != key.end(); ++iterator){
      //Sum all the ASCII values from each character
      myHashCode += (unsigned int) *iterator;
```

```
    }

    //Initialize hash code to 0
    unsigned int finalHash = 0;
    //Store the hash code from the summation result to sumResult
    unsigned sumResult = myHashCode;

    //Iterate through each bit
    for (unsigned int i = 0; i < sizeof(sumResult) * 8; ++i){
        //Store each bit
        unsigned int bit = (sumResult >> i) & 1;
        //Shift hash code by 5 bits to the left and 27 bits to the right
        finalHash = (finalHash << 5) | (finalHash >> 27);
        //Combine the results using XOR operator
        finalHash ^= bit;
    }

    //Use division method for compression function
    unsigned int hashCode = finalHash % capacity;

    //Return the hash code
    return hashCode;
}
```

6. Summation Hashing & Polynomial Hashing Function

This is the hash function that is combined with polynomial and summation hash functions.
How it works is that first I carried out the summation hashing techniques, and during the polynomial hashing phase, I added the has code from the summation each time.

Pseudocode

```
Algorithm: SumPolyHash (key)
Input: string key

    unsigned int myHashCode ← 0

    for each character ch in key do
        myHashCode ← myHashCode + (unsigned int) *iterator

    unsigned int polyHash ← 0
    unsigned int a ← 33
    unsigned int b ← 1

    for each character ch in key do
      polyHash ← polyHash + ((unsigned int) *iterator + myHashCode) * b
      b ← b * a

    unsigned int hashCode ← polyHash modulo capacity

    return hashCode
```

Implementation in C++

```cpp
//Function with summation and polynomial techniques
unsigned int HashTable::SumPolyHash(const std::string& key){
   //Initialize myHashCode to 0
   unsigned int myHashCode = 0;

   //Iterate through each character in the given word
   for(std::string::const_iterator iterator = key.begin(); iterator != key.end(); ++iterator){
      //Sum all the ASCII values from each character
      myHashCode += (unsigned int) *iterator;
   }

   //Initialize polyHash to 0
   unsigned int polyHash = 0;
   //Initialize a to 33
   unsigned int a = 33;
```

```
    //Initialize b to 1
    unsigned int b = 1;

    //Iterate through each character within the given word
    for (std::string::const_iterator iterator = key.begin(); iterator != key.end(); ++iterator){
        //Add hash code to myHashCode and multiply it by b each time
        polyHash += ((unsigned int) *iterator + myHashCode) * b;
        //Iteratively raise the power of a
        b = a * b;
    }

    //Obtain the hash code using division method
    unsigned int hashCode = polyHash % capacity;

    //Return the hash code
    return hashCode;
}
```

Experimental Results

Types of Hash Function:	Types of Compress Function:	The number of Collisions:
Polynomial Hash Function	MAD method	Done! The number of collisions is: 699 The number of unique words is: 24698 The total number of words is: 306569 > count_collisions The number of collisions is: 699 >
Summation Hash Function	Division Method	Done! The number of collisions is: 22945 The number of unique words is: 24698 The total number of words is: 306569 > count_collisions The number of collisions is: 22945 >
Cycle Shift Hash Function	Division Method	Done! The number of collisions is: 667 The number of unique words is: 24698 The total number of words is: 306569 > count_collisions The number of collisions is: 667 >
Polynomial & Cycle Shift Hash Function	**MAD Method**	Done! The number of collisions is: 652 The number of unique words is: 24698 The total number of words is: 306569 > count_collisions The number of collisions is: 652 >

Summation & Cycle Shift Hash Function	Division Method	Done! The number of collisions is: 22949 The number of unique words is: 24698 The total number of words is: 306569 > count_collisions The number of collisions is: 22949 >
Polynomial & Summation Hash Function	MAD Method	Done! The number of collisions is: 663 The number of unique words is: 24698 The total number of words is: 306569 > count_collisions The number of collisions is: 663 >

Analysis of the results:

Based on the experimental results, as shown in the third column, the least number of collisions is **652**, which was obtained from the **Polynomial and Cycle Shift Hash Function using the MAD method** as a compress function. Therefore, this function should be set as the default function to automatically calculate the number of words and collisions.

We observe that the second most efficient hash function in handling collisions is the combination of polynomial and summation hash functions with 663 collisions which are just 11 more collisions than the polynomial and cycle shift hash function. The third most efficient hash function was the cycle shift hash function with 667 collisions. Even though the compression function is a division method, which is less efficient in handling collisions compared to the MAD method, it performed incredibly well.

Overall, as I speculated, the combination of the polynomial and cycle shift hash functions produced the least number of collisions. I assume that the reason for its efficiency in handling collisions is that the benefits of polynomial hashing and cyclic shifting complemented each other in a way that spreaded the words widely throughout the hash table. Specifically, polynomial hashing focuses on order sensitivity, placing words with similar sequences into different indexes by incorporating a constant multiplier. On the other hand, the cyclic shift hash function performs bitwise operations to scramble the bits within the value, increasing the randomness and thereby spreading the words even more effectively across the hash table. Thus, the integration of these functionalities, both of which effectively spread the words, led to an even more efficient way of reducing the number of collisions.

An interesting observation is that I initially thought combining cycle shift hashing techniques with the summation hashing function would significantly improve efficiency of handling collisions compared to

summation hash function, since I thought cyclic shifting would address the unordered aspect of the summation hash function. However, the reality was quite different, as it actually worsened the efficiency of the summation hashing techniques, increasing the number of collisions by 4 more collisions. Therefore, adding cycle shift hashing technique to the summation hash function does not produce any benefits but rather leads to inefficiency in handling collisions.