https://en.wikipedia.org/wiki/List_of_hash_functions (full list of hash functions, some more useful than others)

https://en.wikipedia.org/wiki/Hash_function (need this!!)

https://en.wikipedia.org/wiki/Cyclic_redundancy_check (basically just confusing hash functions)

https://en.wikipedia.org/wiki/Checksum (a bit like a hash function)

https://en.wikipedia.org/wiki/Fingerprint_(computing) (also a bit like a hash function)

https://en.wikipedia.org/wiki/Cryptographic_hash_function (shouldn't need to use cryptographic hash functions as there is little to no risk of hacking and attacks to data)

```
// install dependencies
// using System;
// using System.IO;
// using System.Text;
// using System.Linq;

class BloomFilter
{
   // setup
   protected int[] filter = new int[2682974];

   // used to read the filter from the file and store as an array
   // general use
   public BloomFilter()
   {
      // change filename to the filter.txt
      string filename = "/Users/lewisdrake/Desktop/Bloom Filter/Resources/Filter.txt";
      string text = File.ReadAllText(filename);
      // filter = Array.ConvertAll(tempArray, int.Parse);

      for (uint a = 0; a < filter.Length; a++)t
      {
         filter[a] = text[(int)a] - 48;
      }
   }

   // constructor used to reset the filter
   // initial use
   public BloomFilter(int num)
   {
      for (int a = 0; a < filter.Length; a++)
```

```
        {
            filter[a] = 0;
        }
    }

    // functions
    public bool Lookup(string word)
    {
        if (word.length == 1)
        {
            if (word == "a" || "i")
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        else
        {
            if (filter[Hash1(word)] == 1 && filter[Hash2(word)] == 1 && filter[Hash3(word)] == 1
&& filter[Hash4(word)] == 1 && filter[Hash5(word)] == 1 && filter[Hash6(word)] == 1 &&
filter[Hash7(word)] == 1 && filter[Hash8(word)] == 1 && filter[Hash9(word)] == 1 &&
filter[Hash10(word)] == 1)
            {
                return true;
            }
        }

        return false;
    }

    public void Insert(string word)
    {
        filter[Hash1(word)] = 1;
        filter[Hash2(word)] = 1;
        filter[Hash3(word)] = 1;
        filter[Hash4(word)] = 1;
        filter[Hash5(word)] = 1;
        filter[Hash6(word)] = 1;
        filter[Hash7(word)] = 1;
        filter[Hash8(word)] = 1;
        filter[Hash9(word)] = 1;
        filter[Hash10(word)] = 1;
    }

    public int ViewFilter(int index)
    {
        return filter[index];
    }

    public void WriteFilter()
    {
        string filename = "/Users/lewisdrake/Desktop/Bloom Filter/Resources/Filter.txt";
        string[] stringArray = filter.Select(x => x.ToString()).ToArray();
        string result = String.Concat(stringArray);

        File.WriteAllText(filename, result);
    }

    // hashing
    protected uint CheckSize(uint hash)
    {
        bool loop = true;
        do
        {
            if (hash > filter.Length)
            {
                hash = hash % (uint)filter.Length;
            }
            else
            {
                loop = false;
            }
        } while (loop == true);

        return hash;
    }

    // Pearson Hashing
    protected uint Hash1(string word)
    {
        byte[] nums = { 114, 177, 249, 4, 222, 117, 190, 121, 130, 78, 53, 196, 255, 208, 5, 116,
221, 27, 144, 41, 252, 33, 170, 231, 62, 89, 235, 111, 174, 57, 105, 132, 204, 205, 151, 135,
90, 211, 37, 36, 66, 164, 40, 253, 108, 153, 98, 156, 67, 214, 35, 6, 38, 42, 162, 148, 28, 18,
254, 79, 61, 155, 3, 25, 184, 189, 152, 143, 84, 216, 87, 44, 75, 138, 191, 158, 243, 230, 1,
242, 91, 113, 26, 171, 245, 197, 22, 68, 187, 161, 218, 246, 97, 16, 234, 193, 73, 125, 101,
80, 226, 195, 139, 49, 9, 212, 224, 63, 72, 13, 100, 233, 104, 163, 207, 247, 137, 199, 136,
160, 203, 141, 250, 71, 200, 167, 129, 32, 19, 145, 238, 43, 142, 237, 198, 64, 76, 103, 182,
149, 2, 74, 107, 124, 88, 54, 157, 159, 51, 52, 102, 201, 7, 77, 180, 110, 109, 228, 85, 99,
11, 239, 169, 12, 8, 209, 165, 168, 248, 34, 82, 112, 140, 56, 120, 185, 55, 58, 31, 179, 47,
213, 86, 206, 194, 69, 127, 147, 123, 20, 219, 166, 29, 223, 220, 83, 70, 225, 188, 60, 21,
251, 240, 10, 119, 122, 23, 131, 96, 178, 227, 126, 173, 14, 17, 176, 192, 15, 46, 65, 215,
134, 232, 115, 106, 181, 175, 48, 202, 154, 150, 81, 50, 183, 39, 229, 92, 24, 217, 45, 172,
95, 128, 93, 133, 244, 210, 186, 118, 59, 30, 241, 146, 236, 94 };
```

```
    uint hash = 0;
    uint index;
    byte[] bytes = Encoding.UTF8.GetBytes(word);

    foreach (var a in bytes)
    {
        index = (hash ^ a) % (uint)nums.Length;
        hash = nums[index];
    }

    hash = CheckSize(hash);
    return hash;
}

// Hashing by cyclic polynomial (Buzhash)
protected uint Hash2(string word)
{
    uint hash = 1;

    for (int a = 0; a < word.Length - 1; a++)
    {
        hash = CircularShift(hash) ^ CircularShift((byte)word[a]) ^ (byte)word[a + 1];
    }

    hash = CheckSize(hash);
    return hash;
}

// Fowler-Noll-Vo (FNV-0) hash
protected uint Hash3(string word)
{
    uint hash = 0;
    byte[] bytes = Encoding.UTF8.GetBytes(word);

    foreach (var a in bytes)
    {
        hash = hash * 16777619;
        hash = hash ^ a;
    }

    hash = CheckSize(hash);
    return hash;
}

// dijb2
protected uint Hash4(string word)
{
    uint hash = 5381;
    byte[] bytes = Encoding.UTF8.GetBytes(word);

    foreach (var a in bytes)
    {
        hash = ((hash << 5) + hash) + 33;
    }

    hash = CheckSize(hash);
    return hash;
}

// sdbm
protected uint Hash5(string word)
{
    uint hash = 0;
    byte[] bytes = Encoding.UTF8.GetBytes(word);

    foreach (var a in bytes)
    {
        hash = 65599 + (hash << 6) + (hash << 16) - hash;
    }

    hash = CheckSize(hash);
    return hash;
}

// PJW hash function
protected uint Hash6(string word)
{
    uint hash = 0;
    uint bits = (sizeof(uint) * 8);
    uint max = (uint)(0xFFFFFFFF) << (int)(bits - (bits / 8));

    for (int a = 0; a < word.Length; a++)
    {
        hash = hash << (int)(bits / 8) + word[a];

        if (max != 0)
        {
            hash = hash ^ (max >> (int)bits * 3 / 4) & (~max);
        }
    }

    hash = CheckSize(hash);
    return hash;
}
```

```
// Fast-Hash
protected uint Hash7(string word)
{
    uint a = unchecked((uint)0x880355f21e6d1965);
    uint b = 0;
    uint hash = 144 ^ ((uint)word.Length * a);

    for (uint c = 0; c < word.Length; c++)
    {
        hash ^= Mix(c);
        hash *= a;
    }

    switch (word.Length & 7)
    {
        case 7:
            b ^= (uint)word[6] << 48;
            break;
        case 6:
            b ^= (uint)word[5] << 40;
            break;
        case 5:
            b ^= (uint)word[4] << 32;
            break;
        case 4:
            b ^= (uint)word[3] << 24;
            break;
        case 3:
            b ^= (uint)word[2] << 16;
            break;
        case 2:
            b ^= (uint)word[1] << 8;
            break;
        case 1:
            b ^= (uint)word[0];
            break;
    }

    hash ^= Mix(b);
    hash *= a;
    hash = Mix(hash);

    hash = CheckSize(hash);
    return (uint)hash;
}

// Rabin Fingerprint
protected uint Hash8(string word)
```

```
    {
        uint length = (uint)word.Length;
        uint hash = word[0];

        for (int a = 1; a < word.Length; a++)
        {
            hash += (uint)Math.Pow(word[a] * length, a);
        }

        hash = CheckSize(hash);
        return hash;
    }

    // Fletcher-32
    protected uint Hash9(string word)
    {
        uint a = 0;
        uint b = 0;

        for (int c = 0; c < word.Length; c++)
        {
            a = (a + word[c] % (uint)0xffff);
            b = (a + b) % (uint)0xffff;
        }

        uint hash = (b << 16) | a;

        hash = CheckSize(hash);
        return hash;
    }

    // CRC32
    protected uint Hash10(string word)
    {
        uint hash = 0xffffffff;
        uint tableLookup;
        byte[] nums = { 114, 177, 249, 4, 222, 117, 190, 121, 130, 78, 53, 196, 255, 208, 5, 116,
221, 27, 144, 41, 252, 33, 170, 231, 62, 89, 235, 111, 174, 57, 105, 132, 204, 205, 151, 135,
90, 211, 37, 36, 66, 164, 40, 253, 108, 153, 98, 156, 67, 214, 35, 6, 38, 42, 162, 148, 28, 18,
254, 79, 61, 155, 3, 25, 184, 189, 152, 143, 84, 216, 87, 44, 75, 138, 191, 158, 243, 230, 1,
242, 91, 113, 26, 171, 245, 197, 22, 68, 187, 161, 218, 246, 97, 16, 234, 193, 73, 125, 101,
80, 226, 195, 139, 49, 9, 212, 224, 63, 72, 13, 100, 233, 104, 163, 207, 247, 137, 199, 136,
160, 203, 141, 250, 71, 200, 167, 129, 32, 19, 145, 238, 43, 142, 237, 198, 64, 76, 103, 182,
149, 2, 74, 107, 124, 88, 54, 157, 159, 51, 52, 102, 201, 7, 77, 180, 110, 109, 228, 85, 99,
11, 239, 169, 12, 8, 209, 165, 168, 248, 34, 82, 112, 140, 56, 120, 185, 55, 58, 31, 179, 47,
213, 86, 206, 194, 69, 127, 147, 123, 20, 219, 166, 29, 223, 220, 83, 70, 225, 188, 60, 21,
251, 240, 10, 119, 122, 23, 131, 96, 178, 227, 126, 173, 14, 17, 176, 192, 15, 46, 65, 215,
134, 232, 115, 106, 181, 175, 48, 202, 154, 150, 81, 50, 183, 39, 229, 92, 24, 217, 45, 172,
95, 128, 93, 133, 244, 210, 186, 118, 59, 30, 241, 146, 236, 94, 0 };

        byte[] bytes = Encoding.UTF8.GetBytes(word);

        foreach (var b in bytes)
        {
            tableLookup = (hash ^ b) & 0xff;
            tableLookup = tableLookup % (uint)nums.Length;
            hash = (hash >> 8) ^ nums[tableLookup];
        }

        hash = hash ^ 0xffffffff;

        hash = CheckSize(hash);
        return hash;
    }

    // Helpers
    // Hash2
    protected uint CircularShift(uint a)
    {
        uint b = a << 1 | a >> 31;
        return b;
    }

    // Hash7
    protected uint Mix(uint hash)
    {
        long a = (long)hash;
        a ^= a >> 23;
        a *= 0x2127599bf4325c37;
        a ^= a >> 47;

        hash = (uint)a;
        return (uint)hash;
    }
}
```