Hash Functions

Strings as keys

- Keys are not always integers.

- The classic example is a dictionary. If you want to put every word of an English-language dictionary, from a to zyzzyva
  - (who knows what it is and how to pronounce it?)

into your computer's memory so they can be accessed quickly, a hash table is a good choice.

Strings as keys

- Keys are not always integers.

- A similar widely used application for hash tables is in computer-language compilers, which maintain a symbol table in a hash table.

- The symbol table holds all the variable and function names made up by the programmer, along with the address where they can be found in memory.
    - https://en.wikipedia.org/wiki/Symbol_table

Converting words to numbers

- Let’s say we want to store **a 50,000**-word English-language dictionary in main memory

Use the length of a word as a hash function:

A: It is a good choice

B: It is not uniform

C: It is not fast

D: It is not deterministic

Converting words to numbers: 50,000 words

- Another idea: Use ASCII code, in which *a* is 97, *b* is 98, and so on, up to 122 for *z.* Then **add** the numbers.
- Let's say *a* is 1, *b* is 2, *c* is 3, and so on up to 26 for *z.*
- We'll also say a blank is 0, so we have 27 characters. (assume no capitals).
  - *I subtracted 96 to make the math easier*
- "cat"= 3 + 1 + 20 = 24.

Converting words to numbers: 50,000 words

- Another idea: Use ASCII code, in which *a* is 97, *b* is 98, and so on, up to 122 for *z.* Then **add** the numbers.
- Let's say *a* is 1, *b* is 2, *c* is 3, and so on up to 26 for *z.*
- We'll also say a blank is 0, so we have 27 characters. (assume no capitals).
  - *I subtracted 96 to make the math easier*
- "cat"= 3 + 1 + 20 = 24.

Use the ASCII as a hash function:

A: It is a good choice

B: It is not uniform

C: It is not fast

D: It is not deterministic

Converting words to numbers: multiplying powers

- Idea: Hash each word into a unique location.
- How? Number analogy:

$7{,}546 = 7*1000 + 5*100 + 4*10 + 6*1$

(or using powers of 10):

$7{,}546 = 7*10^3 + 5*10^2 + 4*10^1 + 6*10^0$

Why base 10? Because there are **10** possible digits.

Converting words to numbers: multiplying powers

Why base 10? Because there are **10** possible digits.

If we are apply the same idea to strings, what is our base?

A: 10

B: 27 (26 letters and the space)

C: Can’t apply the method

Converting words to numbers: multiplying powers

- We will use base 27 because we have 26 letters and 1 space.
- “cat”: c = 3, a = 1, t = 20.

What is the corresponding conversion?

A: $3*27^3 + 1*27^2 + 20*27^1$

B: $3*27^2 + 1*27^1 + 20*27^0$

C: $3*27 + 1*27 + 20*27$

Converting words to numbers: multiplying powers

The largest 10-letter word (let's assume 10 is the max length of the word), *zzzzzzzzzz,* translates into …

?

Converting words to numbers: multiplying powers

The largest 10-letter word (let's assume 10 is the max length of the word), *zzzzzzzzzz,* translates into …

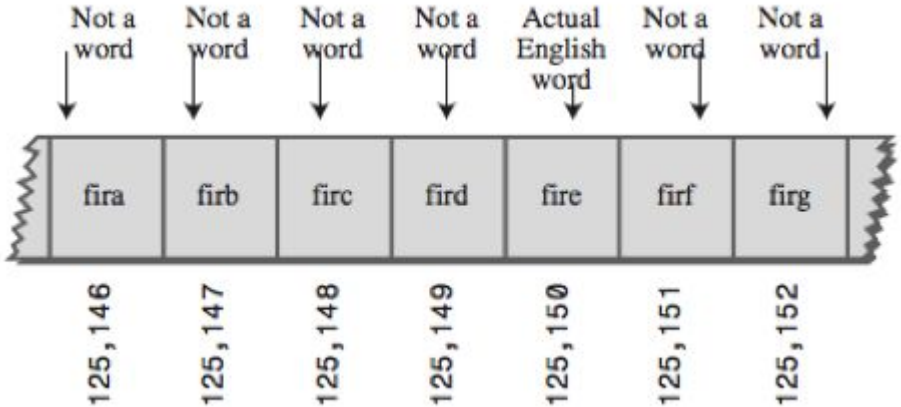
$26*27^9 + 26*27^8 + 26*27^7 + 26*27^6 + 26*27^5 + 26*27^4 + 26*27^3 + 26*27^2 + 26*27^1 + 26*27^0$ <-HUGE NUMBER.

Converting words to numbers: multiplying powers

The largest 10-letter word (let's assume 10 is the max length of the word), *zzzzzzzzzz*, translates into …

$26*27^9 + 26*27^8 + 26*27^7 + 26*27^6 + 26*27^5 + 26*27^4 + 26*27^3 + 26*27^2 + 26*27^1 + 26*27^0$ <-HUGE NUMBER.

Another problem:



Conclusion

- Our first scheme—adding the numbers—generated too few indices.
- This latest scheme—adding the numbers times powers of 27—generates too many.

Conclusion

- Our first scheme—adding the numbers—generated too few indices.
- This latest scheme—adding the numbers times powers of 27—generates too many.
- Hash Tables have large but reasonable size. We will use the second approach with some changes.

$$key = 3*27^2 + 1*27^1 + 20*27^0$$

```
index = (key) % tableSize;
```

Hash Strings: “cat”

hashVal= $3*27^2 + 1*27^1 + 20*27^0$

```
public static int hashFunc1(String key)
   {
   int hashVal = 0;
   int pow27 = 1;                          // 1, 27, 27*27, etc

   for(int j=key.length()-1; j>=0; j--) // right to left
      {
      int letter = key.charAt(j) - 96;  // get char code
      hashVal += pow27 * letter;        // times power of 27
      pow27 *= 27;                      // next power of 27
      }
   return hashVal % arraySize;
   }  // end hashFunc1()
```

Not efficient yet

Aside from the character conversion, there are **two** multiplications and an **addition** inside the loop.

- **Horner's Method.**

$$var4*n^4 + var3*n^3 + var2*n^2 + var1*n^1 + var0*n^0$$

can be written as

$$(((var4*n + var3)*n + var2)*n + var1)*n + var0$$

Another version

```
public static int hashFunc2(String key)
      {
      int hashVal = key.charAt(0) - 96;

      for(int j=1; j<key.length(); j++)    // left to right
         {
         int letter = key.charAt(j) - 96;  // get char code
         hashVal = hashVal * 27 + letter;  // multiply and add
         }
      return hashVal % arraySize;          // mod
      }  // end hashFunc2()
```

Are we done improving?

A: Yes, looks good.

B: No, it is not deterministic

C: No, it is not uniform

D: No, not fast

E: No, can’t handle large words

Final modification: (do not -96 in your HW)

```
public static int hashFunc3(String key)
   {
   int hashVal = 0;
   for(int j=0; j<key.length(); j++)    // left to right
      {
      int letter = key.charAt(j) - 96;  // get char code
      hashVal = (hashVal * 27 + letter) % arraySize; // mod
      }
   return hashVal;                      // no mod
   }  // end hashFunc3()
```

Even faster

Various bit- manipulation tricks can be played as well, such as using a base of 32 (or a larger power of 2) instead of 27, so that multiplication can be effected using the shift operator (>>)

Right and Left shifts

- https://en.wikipedia.org/wiki/Logical_shift
- <<, >>> is logical shift right.
- >> is arithmetic shift right: the sign bit is extended to preserve the signedness of the number.

CRC (simplified)
Cyclical Redundancy Checking (may have to use **abs**)

```
public static int hashString(String str)
{
   int hashValue =0;
   for (int i=0; i<str.length(); i++)
   {
     int leftShiftedValue = hashValue <<5; //left shift

     int rightShiftedValue = hashValue >>>27; //right shift

      // | is bitwise OR,  ^ is bitwise XOR
      hashValue = (leftShiftedValue | rightShiftedValue) ^ str.charAt(i);
    }
    return hashValue % M;
}
```

Cyclical Redundancy Checking: help in error detection

- CRC is a hash function designed to detect accidental changes to raw computer data.
- It is commonly used in digital networks and storage devices such as hard disk drives.
- A CRC-enabled device calculates a short, fixed-length binary sequence, known as the CRC code, for each block of data and sends or stores them both together.
- When a block is read or received the device repeats the calculation; if the new CRC code does not match the one calculated earlier, then the block contains a data error and the device may take corrective action such as requesting the block be sent again. wiki