Hash Functions

• Why are hash tables good?

Hash ADT

• Transforms a key into a location where we store a value
• Two primary parts of hashing
  o Hash Function: converts the key into a location
  o Collision Resolution: How to handle the problem of multiple keys hashing to the same location

Hash Functions

• Purpose: transforms the keys into table addresses
  o Arithmetic computation
  o Simple to implement
  o Produces an index in the range \([0, M-1]\) assuming there are \(M\) addresses in the table
• Example hash function

\[
\text{int hash (KeyType key) \{}
\]
\[
\quad \text{return 0;}
\]
\[
\}
\]

  o What’s wrong?

  • Another Hash Function

\[
\text{int hash (KeyType key, int tableSize) \{}
\]
\[
\quad \text{int keyInt = (int) key;}
\]
\[
\quad \text{return keyInt % tableSize;}
\]
\[
\}
\]

  o When is this good?

Hashing Keys from Strings

• Assume that we are hashing 3 letter words
  o Use the ASCII codes and treat them as 128 base numbers
    ▪ e.g. “now” = 110 * 128^2 + 111 * 128^1 + 119 * 128^0
    ▪ pow function is expensive
    ▪ Is there an alternative?

  “now” =
Hash Functions for Long Keys

- Transform keys piece by piece
- Use Horner’s Rule and cast out multiples at each step

```c
int hash(char *key, int tableSize) {
    int hash = 0, alpha = 127;
    for (; *key != 0; key++)   // 0 == NULL character
        hash = (alpha*hash + (*key)) % tableSize;
    return hash;
}
```

- Set a to 127 instead of 128 to avoid anomalies if table size is a power or multiple of 2

Universal Hash Function of String Keys

- Don’t need to stick to the letter of the definition of modular hashing
- Goal: Involve all the bits of the key in a computation that produces an integer less then M
- Theoretical ideal: chance of collision between two distinct keys is precisely 1/M
- Use pseudorandom coefficient values instead of a fixed radix

```c
int hashU (char *key, int tableSize) {
    int hash, alpha = 31415, beta = 27183;
    for (hash = 0; *key != 0;
         key++, alpha = alpha*beta % (tableSize-1))
        hash = (alpha*hash + *key) % tableSize;
    return (hash < 0) ? (hash+tableSize) : hash;
}
```

Improving efficiency

- Why should you care?
- Universal hashing may be too slow
- Alternative

```c
int hash (Key key, int tableSize) {
    // tableSize is a power of 2
    return key & (tableSize-1);
}
```

- Use the Chi-squared statistic to test random values