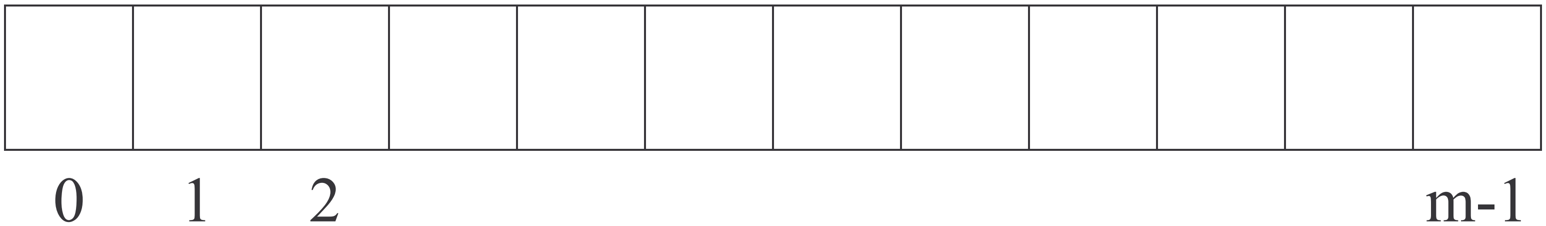
CMSC 341

Hashing

The Basic Problem

- We have lots of data to store.
- We desire efficient – O(1) – performance for insertion, deletion and searching.
- Too much (wasted) memory is required if we use an array indexed by the data's key.
- The solution is a "hash table".

Hash Table



Basic Idea

- The hash table is an array of size ‘m’
- The storage index for an item determined by a *hash function*

  $h(k): U \rightarrow \{0, 1, \ldots, m-1\}$

Desired Properties of h(k)

- easy to compute
- uniform distribution of keys over $\{0, 1, \ldots, m-1\}$
  - when $h(k_1) = h(k_2)$ for $k_1, k_2 \in U$, we have a *collision*

Division Method

The hash function:

$$h(k) = k \bmod m$$

where m is the table size.

m must be chosen to spread keys evenly.

- Poor choice: m = a power of 10
- Poor choice: $m = 2^b$, $b > 1$

A good choice of m is a prime number.

Also we want the table to be no more than 80% full.

- Choose m as smallest prime number greater than m_{min}, where m_{min} = (expected number of entries)/0.8

Multiplication Method

The hash function

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

where A is some real positive constant.

A very good choice of A is the inverse of the "golden ratio."

Given two positive numbers x and y, the ratio x/y is the "golden ratio" if

$$\phi = x/y = (x+y)/x$$

The golden ratio:

$$x^2 - xy - y^2 = 0 \quad \Rightarrow \quad \phi^2 - \phi - 1 = 0$$

$$\phi = (1 + \text{sqrt}(5))/2 \quad = \quad 1.618033989\ldots$$

$$\sim= Fib_i/Fib_{i-1}$$

Multiplication Method (cont.)

Because of the relationship of the golden ratio to Fibonacci numbers, this particular value of A in the multiplication method is called “Fibonacci hashing.”

Some values of

$h(k) = \lfloor m(k\,\phi^{-1} - \lfloor k\,\phi^{-1} \rfloor) \rfloor$

$= 0$ for $k = 0$

$= 0.618m$ for $k = 1$ ($\phi^{-1} = 1/\,1.618\ldots = 0.618\ldots$)

$= 0.236m$ for $k = 2$

$= 0.854m$ for $k = 3$

$= 0.472m$ for $k = 4$

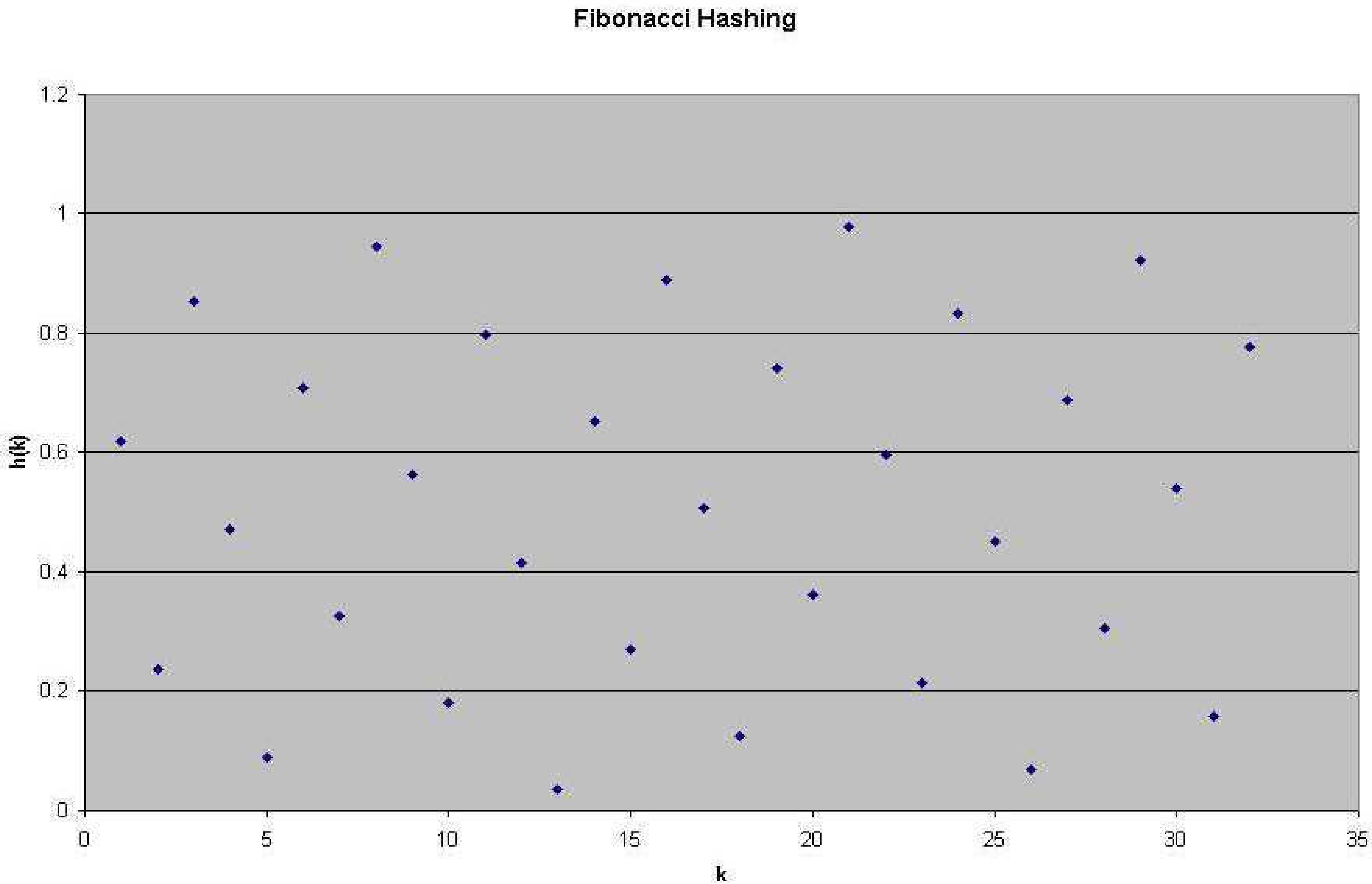
$= 0.090m$ for $k = 5$

$= 0.708m$ for $k = 6$

$= 0.326m$ for $k = 7$

$= \ldots$

$= 0.777m$ for $k = 32$



Fibonacci Hashing
h(k)
1.2
1
0.8
0.6
0.4
0.2
0
0
5
10
15
20
25
30
35
k

Non-integer Keys

In order to has a non-integer key, must first convert to a positive integer:

$$h(k) = g(f(k)) \quad \text{with} \quad f: U \rightarrow \text{integer}, \quad g: I \rightarrow \{0 \,..\, m-1\}$$

Suppose the keys are strings.

How can we convert a string (or characters) into an integer value?

Horner’s Rule

```
int hash(const string &key, int tablesize)
{
  int hashval = 0;

  // f(k) by Horner’s rule
  for (int i = 0; i < key.length(); i++)
      hashval = 37 * hashval + key[i];

   // g(k)  by division method
   hashval %= tablesize;
   if (hashval < 0)              // overflow
     hashval += tablesize;

   return hashval;
}
```

HashTable Class

```
template <typename HashedObj>
class HashTable {
   public:
        explicit HashTable( int size = 101 );
        bool contains( const HashedObj& x ) const;
        void makeEmpty();
        void insert (const HashedObj &x);
        void remove (const HashedObj &x);

   private:
        vector< xxxx > theTable;  // more later
        int currentSize;
        void rehash( );
        int myhash( const HashedObj& x ) const;
}
int hash (int key);
int hash ( const string& key );
```

Hash Table Ops

`bool contains( const HashedObj &x ) const;`

- returns true if x is present in the table

`void insert (const HashedObj &x);`

- if x already in table, do nothing.
- otherwise insert it, using the appropriate hash function

`void remove (const HashedObj &x);`

- remove the instance of x, if x is present
- otherwise, does nothing

`void makeEmpty();`

Hash Functions

```
int myhash( const HashedObject& x ) const
{
  // call user-supplied overloaded hash function
  int hashVal = hash( x );
  hashVal %= theTable.size( );
  if ( hashVal < 0 )
     hashVal += theTable.size( );

  return hashVal;
}
```

Handling Collisions

Collisions are inevitable. How to handle them?

S***eparate chaining hash tables***

- store colliding items in a list
- if m is large enough, list lengths are small

Insertion of key k

- hash(k) to find the proper list
- if k is in that list, do nothing. Else, insert k on that list.

Asymptotic performance

- if always inserted at head of list, and no duplicates, insert = O(1): best, worst, average

Hash Class for Separate Chaining

To implement separate chaining, the private data of the hash table is a vector (array) of Lists. The hash functions are written using List functions

```
private:
    vector< List< HashObj > > theTable;
```

Performance of contains( )

contains

- hash k to find the proper list
- Call contains( ) on that list which returns a boolean

Performance

- best:
- worst:
- average

Performance of remove( )

Remove k from table

- hash k to find proper list
- remove k from list

Performance

- best
- worst
- average

Handling Collisions Revisited

Probing hash tables

- all elements stored in the table itself (so table should be large. Rule of thumb: $m >= 2N$)
- upon collision, item is hashed to a new (open) slot.

Hash function

$h: U \times \{0,1,2,\ldots\} \rightarrow \{0,1,\ldots,m-1\}$

$h(k, i) = (h'(k) + f(i)) \bmod m$

for some $h': U \rightarrow \{0, 1,\ldots, m-1\}$

and some $f(i)$ such that $f(0) = 0$

Each attempt to find an open slot (i.e. calculating $h(k, i)$) is called a ***probe***

Hash Class for Probing Hash Tables

In this case, the hash table is just an array

```
private:
     vector< HashedObj > theTable;
```

Which is allocated space in by the hash table constructor

```
HashTable( int size )
  : theTable( size )
{
  // other constructor code
}
```

Linear Probing

Use a linear function for $f(i)$

$f(i) = c * i$

Example:

$h'(k) = k \bmod 10$ in a table of size 10, $f(i) = i$

So that

$h(k, i) = (k \bmod 10 + i) \bmod 10$

Insert the values U={89,18,49,58,69} into the hash table

Linear Probing (cont'd)

Problem: Clustering

- when the table starts to fill up, performance $\rightarrow O(N)$

Asymptotic Performance

- insertion and unsuccessful find, average
  - λ is the "load factor" – what fraction of the table is used
  - Number of probes $\cong (\frac{1}{2})(1+1/(1-\lambda)^2)$
  - if $\lambda \cong 1$, the denominator goes to zero and the number of probes goes to infinity

Linear Probing (cont'd)

Remove

- Can't just use the hash function(s) to find the object and remove it, because objects that were inserted after X were hashed based on X's presence.

- Can just mark the cell as deleted so it won't be found anymore.
  - Other elements still in right cells
  - Table can fill with lots of deleted junk

Quadratic Probing

Use a quadratic function for $f(i)$

$f(i) = c_2 i^2 + c_1 i + c_0$

The simplest quadratic function is $f(i) = i^2$

Example:

Let $f(i) = i^2$ and $m = 10$

Let $h'(k) = k \bmod 10$

So that

$h(k, i) = (k \bmod 10 + i^2) \bmod 10$

Insert the value U={89, 18, 49, 58, 69 } into an initially empty hash table

Quadratic Probing (cont.)

Advantage:

- reduced clustering problem

Disadvantages:

- reduced number of sequences
- no guarantee that empty slot will be found if $\lambda \geq 0.5$, even if m is prime
- If m is not prime, may not find an empty slot even if $\lambda < 0.5$

Double Hashing

Let f(i) use another hash function

$f(i) = i * h_2(k)$

Then $h(k, I) = (h'(k) + \; * h_2(k)) \bmod m$

And probes are performed at distances of

$h_2(k), 2 * h_2(k), 3 * h_2(k), 4 * h_2(k)$, etc

Choosing $h_2(k)$

- don't allow $h_2(k) = 0$ for any k.
- a good choice:
  $h_2(k) = R - (k \bmod R)$ with R a prime smaller than m

Characteristics

- No clustering problem
- Requires a second hash function

Rehashing

If the table gets too full, the running time of the basic operations starts to degrade.

For hash tables with separate chaining, “too full” means more than one element per list (on average)

For probing hash tables, “too full” is determined as an arbitrary value of the load factor.

To rehash, make a copy of the hash table, double the table size, and insert all elements (from the copy) of the old table into the new table

Rehashing is expensive, but occurs very infrequently.