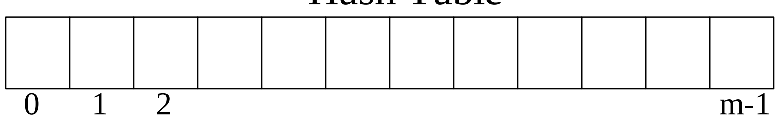
CMSC 341
Lecture 12

## Hash Table




Basic Idea

- an array in which items are stored
- 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, …, m}
  - when $h(k_1) = h(k_2)$ for $k_1, k_2 \in U$ , we have a *collision*

## Division Method

The function:

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

where m is the table size.

M must be chosen to spread keys evenly.

- Ex: m = a factor of 10
- Ex: $m = 2^b$, 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 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 rule."

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:

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

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

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

## 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… = 0.618…)

= 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

= …

= 0.777m for k = 32

## Non-integer Keys

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

$h(k) = g(f(k))$ with $f: U \rightarrow int$

$g: I \rightarrow \{0 .. m-1\}/2$

Suppose the keys are strings. How can we convert a string (or characters) into an integer value?

```
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*hasval + key[i];

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

## HashTable Class

```
template <class HashedObj>
class HashTable {
public:
   explicit HashTable(const HashedObj) &notFound, size=101);
   HashTable(const HashTable &rhs) :
   ITEM_NOT_FOUND(rhs.ITEM_NOT_FOUND),theLists(rhs.theLists){
   }
   const HashedObj &find(const HashedObj &x) const;
   void makeEmpty();
   void insert (const HashedObj &x);
   void remove (const HashedObj &x);
   const HashTable &operator=(const HashTable &rhs);
private:
   vector<List<HashedObj>> theLists;
   const HashedObj ITEM_NOT_FOUND;
};
```

## Hash Table Ops

```
const HashedObj &find(const HashedObj &x)
  const;
```

- returns the HashedObj in the table, if present
- otherwise, returns ITEM_NOT_FOUND

```
void insert (const HashedObj &x);
```

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

```
void remove (const HashedObj &x);
```

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

```
void makeEmpty();
```

## Handling Collisions

Collisions are inevitable. How to handle them?

One possibility: *separate chaining* (aka *open hashing*)

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

Insertion of key k

- hash(k) to find bucket
- if k is on that this, 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

## Find Performance

Find

- hash k to find the bucket
- do a find on that list, returns a listItr
- if itr.isPastEnd(), return ITEM_NOT_FOUND, otherwise, return itr.retrieve()

Performance

- best:
- worst:
- average

## Remove Performance

Remove k from table

- hash k to find bucket
- remove k from list

Performance

- best
- worst
- average