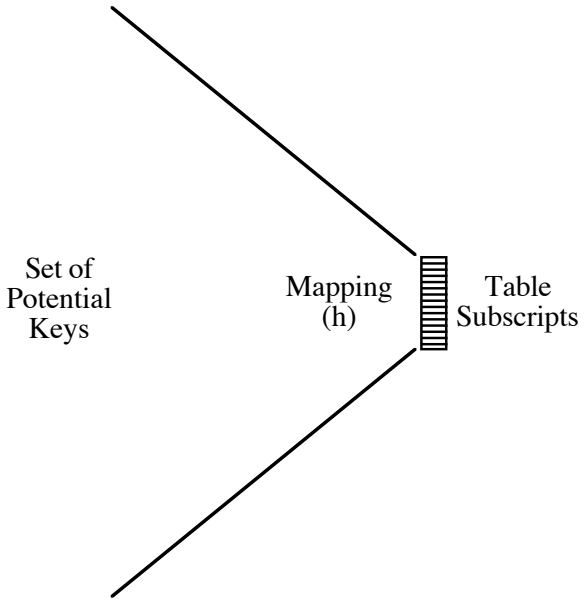
CSE 2320 Notes 11: Hashing

(Last updated 10/19/06 8:17 PM)

CLRS, 11.1-11.4 (skip 11.3.3)

CONCEPTS

Goal: Achieve faster (nearly O(1) time) operations than balanced trees by using "randomness" in key sets by sacrificing 1) generality and 2) ordered retrieval.



Regardless of the hash function, a dynamic set of keys will lead to *collisions*.

Birthday paradox

366 different birthdays available

How many (random) persons are needed to have at least even odds of two persons with the *same* birthday? 23

Probability of k persons having k *different* birthdays is $\prod_{i=1}^{k-1} \frac{366 - i}{366}$

```
probability of unique birthdays among 0 people is 1
probability of unique birthdays among 1 people is 1
probability of unique birthdays among 2 people is 0.997268
probability of unique birthdays among 3 people is 0.991818
probability of unique birthdays among 21 people is 0.557221
probability of unique birthdays among 22 people is 0.525249
probability of unique birthdays among 23 people is 0.493677
probability of unique birthdays among 24 people is 0.462654
probability of unique birthdays among 57 people is 0.0100102
probability of unique birthdays among 58 people is 0.00845124
```

HASH FUNCTIONS

Remaindering (division method)

h(key) = key % m

m is the table size

Folklore: Make m prime, regardless of collision handling technique. Double hashing requires.

```
int nextPrime(int x)
{
int work,k,remainder,quotient;

if (x%2==1)
  work=x;
else
  work=x+1;

while (1)
{
  for (k=3; ;k+=2)
  {
    remainder=work%k;
    if (remainder==0)
      break;
    quotient=work/k;
    if (quotient<k)
      return work;
  }

  work+=2;
}
}
```

Multiplication

```
hash = m * (0.710123587*key - (int)(0.710123587*key));
```

Universal Hashing - aside

Use parameterized hash function to minimize chance of getting collisions beyond expectation.

Parameter is randomly generated when hash structure is initialized.

Text Strings as Key

```
scanf("%s",str);
hash=0;
for (i=0;
     str[i]!=0;
     i++)
  hash = (hash*10 + str[i]) % m;
printf("%s => %d\n",str,hash);
```

A string's *signature* may be stored in a data structure, even if hashing is not used.

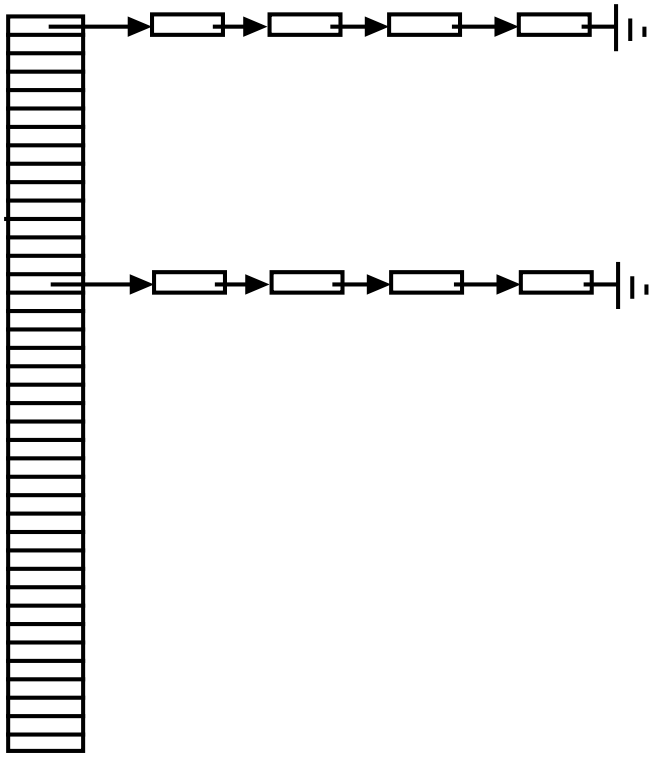
COLLISION HANDLING BY CHAINING

Concept – Use table of pointers to unordered linked lists. Elements of a list have the same signature.

Load Factor (α) = $\frac{\#\text{ elements stored}}{\#\text{ slots in table}}$

Often stated as a per cent. For some methods, such as chaining, α can exceed 100%.

Expected probes is $\frac{n}{2m} = \frac{\alpha}{2}$ for hits and $\frac{n}{m} = \alpha$ for misses.



COLLISION HANDLING BY OPEN ADDRESSING

Saves space when records are small and chaining would waste a large fraction of space for links.

Collisions are handled by using a *probe sequence* for each key – a permutation of the table's subscripts.

Hash function is h(key, i) where i is the number of reprobe attempts tried.

Two special key values (or flags) are used: *never-used* (-1) and *recycled* (-2). Searches stop on *never-used*, but continue on *recycled*.

Linear Probing - h(key, i) = (key + i) % m

Properties:

1. Probe sequences eventually hit all slots.
2. Probe sequences wrap back to beginning of table.
3. Exhibits lots of *primary clustering* (the end of a probe sequence coincides with another probe sequence):

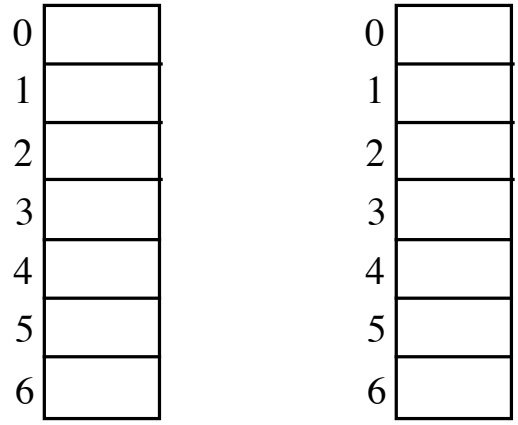
   $i_0\ i_1\ i_2\ i_3\ i_4 \ldots\ i_j\ i_{j+1} \ldots$
   $i_j\ i_{j+1}\ i_{j+2} \ldots$

4. There are only m probe sequences.

What about using h(key, i) = (key + 2*i) % 101 or h(key, i) = (key + 50*i) % 1000?

Suppose all keys are *equally likely* to be accessed. Is there a best order for inserting keys?

Insert keys: 101, 171, 102, 103, 104, 105, 106



Quadratic Probing (historical aside) – $h(key, i) = (key + c_1 i + c_2 i^2) \% m$ (usually $c_1 = c_2 = 1$)

Properties:

1. Probe sequences guaranteed to eventually hit only half of the slots.
2. Still only m different probe sequences.
3. Eliminates most primary clustering, but not *secondary clustering*: if two keys have the same initial probe, then their probe sequences are the same. (Also occurs for linear probing.)

Double Hashing – $h(key, i) = (h_1(key) + i*h_2(key)) \% m$

Properties:

1. Probe sequences will hit all slots only if m is prime.
2. $m*(m - 1)$ probe sequences.
3. Eliminates most clustering.

Hash Functions:

$h_1 = key \% m$

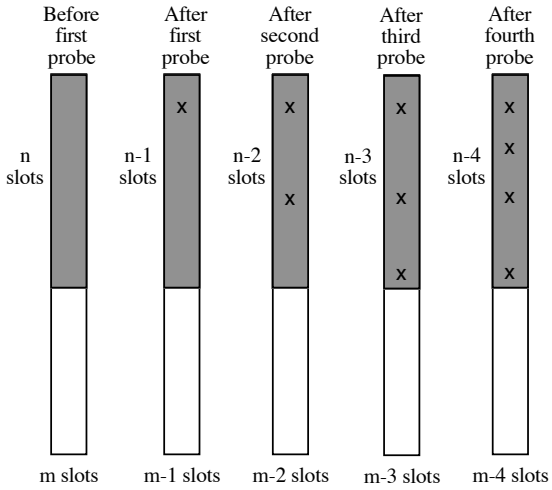
a. $h_2 = 1 + key \% (m - 1)$

b. $h_2 = 1 + (key/m) \% (m - 1)$

c. Use last few bits of key as h_2, but must avoid zero.

Double hashing comes very close to these results, but analysis assumes that hash function provides all m! permutations of subscripts.

1. Unsuccessful search with load factor of $\alpha = \frac{n}{m}$. Each successive probe has the effect of decreasing table size and number of slots in use by one.



   a. Probability that all searches have a first probe 1

   b. Probability that search goes on to a second probe $\alpha = \frac{n}{m}$

   c. Probability that search goes on to a third probe $\alpha \frac{n-1}{m-1} < \alpha \frac{n}{m} < \alpha^2$

   d. Probability that search goes on to a fourth probe $\alpha \frac{n-1}{m-1} \frac{n-2}{m-2} < \alpha^2 \frac{n-2}{m-2} < \alpha^3$

   . . .

   Suppose the table is large. Sum the probabilities for probes to get upper bound on expected number of probes:

   $$\sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$ (much worse than chaining)

2. Inserting a key with load factor α

   a. Exactly like unsuccessful search

   b. $\frac{1}{1-\alpha}$ probes

3. Successful search

   a. Searching for a key takes as many probes as inserting *that particular key*.

   b. Each inserted key increases the load factor, so the inserted key number i + 1 is expected to take no more than

$$\frac{1}{1-\frac{i}{m}} = \frac{m}{m-i} \text{ probes}$$

   c. Find expected probes for n consecutively inserted keys (each key is equally likely to be requested):

$$\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} = \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \qquad \text{Sum is } \frac{1}{m}+\frac{1}{m-1}+\ldots+\frac{1}{m-n+1}$$

$$= \frac{m}{n}\sum_{i=m-n+1}^{m}\frac{1}{i}$$

$$\le \frac{m}{n}\int_{m-n}^{m}\frac{1}{x}dx \qquad \text{Upper bound on sum for decreasing function. CLRS, p. 1067 (A.12)}$$

$$= \frac{m}{n}\left(\ln m - \ln(m-n)\right) = \frac{1}{\alpha}\ln\frac{m}{m-n} = \frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

```
alpha 0.200 unsuccessful (insert) 1.250 successful 1.116
alpha 0.250 unsuccessful (insert) 1.333 successful 1.151
alpha 0.300 unsuccessful (insert) 1.429 successful 1.189
alpha 0.350 unsuccessful (insert) 1.538 successful 1.231
alpha 0.400 unsuccessful (insert) 1.667 successful 1.277
alpha 0.450 unsuccessful (insert) 1.818 successful 1.329
alpha 0.500 unsuccessful (insert) 2.000 successful 1.386
alpha 0.550 unsuccessful (insert) 2.222 successful 1.452
alpha 0.600 unsuccessful (insert) 2.500 successful 1.527
alpha 0.650 unsuccessful (insert) 2.857 successful 1.615
alpha 0.700 unsuccessful (insert) 3.333 successful 1.720
alpha 0.750 unsuccessful (insert) 4.000 successful 1.848
alpha 0.800 unsuccessful (insert) 5.000 successful 2.012
alpha 0.850 unsuccessful (insert) 6.667 successful 2.232
alpha 0.900 unsuccessful (insert) 10.000 successful 2.558
alpha 0.910 unsuccessful (insert) 11.111 successful 2.646
alpha 0.920 unsuccessful (insert) 12.500 successful 2.745
alpha 0.930 unsuccessful (insert) 14.286 successful 2.859
alpha 0.940 unsuccessful (insert) 16.666 successful 2.993
alpha 0.950 unsuccessful (insert) 20.000 successful 3.153
alpha 0.960 unsuccessful (insert) 25.000 successful 3.353
alpha 0.970 unsuccessful (insert) 33.333 successful 3.615
alpha 0.980 unsuccessful (insert) 49.998 successful 3.992
alpha 0.990 unsuccessful (insert) 99.993 successful 4.652
```