Hash Tables

Complexity using recurrence

- **Consider an algorithm which is recursive and assume it takes T(n) as the complexity**
- **For binary search, we divide into two equal parts and we work on one of the**
- **Assume the it takes f(n)=O(1) to combine the results**
- **Then, T(n)=T(n/2)+f(n)**
- **If f(n)=1, then**
- **$T(n) = 1 + T(n/2) = 1 + (1 + T(n/4)) = 2 + T(n/4)$ $= 2 + (1 + T(n/8)) = 3 + T(n/8) = \ldots$ $k + T(n/2^k) = \log n + T(n/2^{\log n}) = \log n + T(1) =$ $\log n + 1 = O(\log n)$.**

Master theorem for complexity

- **General formula that works if recurrence has the form**
- $T(n) = aT(n/b) + f(n)$
    - ***a*** **is number of subproblems**
    - ***n/b*** **is size of each subproblem**
    - ***f(n)*** **is cost of non-recursive part**

a) If $f(n) = O(n^{\log_b(a)-\varepsilon})$, then $T(n)=\Theta(n^{\log_b(a)})$.

b) If $f(n) = \Theta(n^{\log_b(a)})$, then $T(n)=\Theta(n^{\log_b(a)} \log(n))$.

c) If $f(n) = \Omega(n^{\log_b(a)+\varepsilon})$ and then $T(n)=\Theta(f(n))$

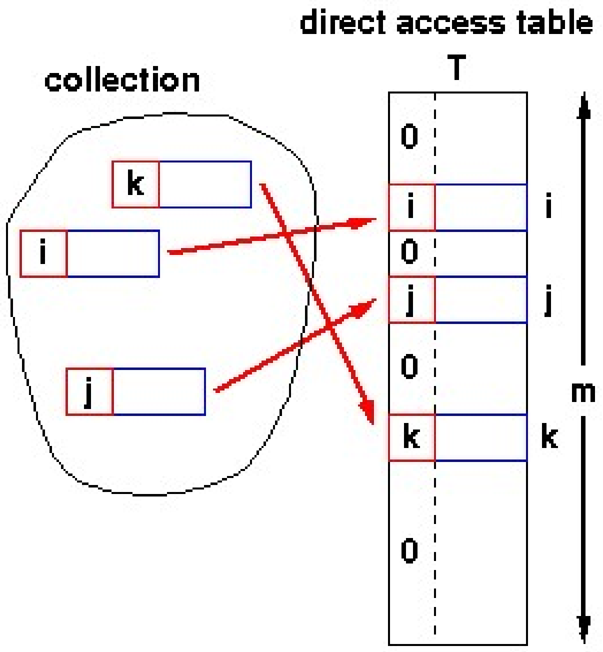
Hash Tables

- All search structures so far
  - Relied on a comparison operation
  - Performance $O(n)$ or $O(\log n)$
- Assume I have a function
  - $f(\,key\,) \rightarrow integer$

    ie one that maps a key to an integer
- What performance might I expect now?

Hash Tables - Structure

- **Simplest case:**
  - Assume items have integer keys in the range $1 .. m$
  - Use the value of the key itself to select a slot in a direct access table in which to store the item
  - To search for an item with key, k, just look in slot k
    - If there's an item there, you've found it
    - If the tag is 0, it's missing.
  - Constant time, $O(1)$



Hash Tables - Constraints

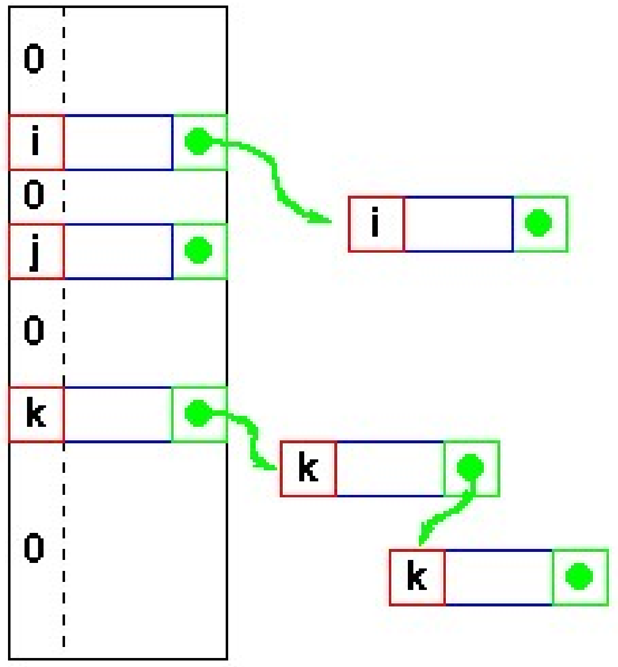
- **Constraints**
  - **Keys must be unique**
  - **Keys must lie in a small range**
  - **For storage efficiency,
    keys must be dense in the range**
  - **If they're sparse (lots of gaps between values),
    a lot of space is used to obtain speed**
    - **Space for speed trade-off**

Hash Tables - Relaxing the constraints

- **Keys must be unique**
  - Construct a linked list of duplicates "attached" to each slot
  - If a search can be satisfied by *any* item with key, k, performance is still $O(1)$

    but
  - If the item has some other distinguishing feature which must be matched, we get $O(n^{max})$

    where n^{max} is the largest number of duplicates - or length of the longest chain

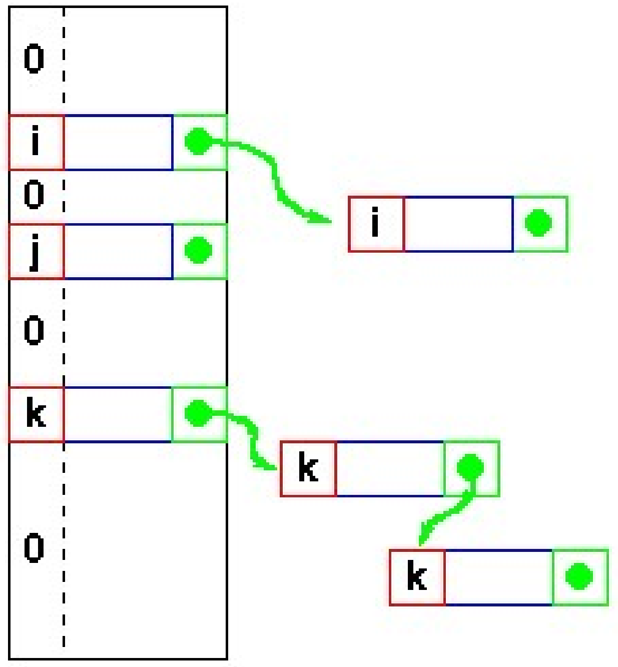


Hash Tables - Relaxing the constraints

- **Keys are integers**
  - **Need a hash function**

    $h(\mathit{key}) \rightarrow \mathit{integer}$

    ie **one that maps a key to an integer**
  - **Applying this function to the key produces an address**
  - **If h maps each key to a *unique integer* in the range $0 .. m\text{-}1$ then search is $O(1)$**



Hash Tables - Hash functions

- **Form of the hash function**
  - Example - using an *n*-character key

    ```
    int hash( char *s, int n ) {
        int sum = 0;
        while( n-- ) sum = sum + *s++;
        return sum % 256;
        }
    ```

    returns a value in **0 .. 255**
  - `xor` function is also commonly used

    ```
        sum = sum ^ *s++;
    ```
  - But any function that generates integers in **0..*m*-1** for some suitable (*not too large*) ***m*** will do
  - As long as the hash function itself is ***O*(1)** !

Hash Tables - Collisions

- **Hash function**
  - **With this hash function**

```
int hash( char *s, int n ) {
    int sum = 0;
    while( n-- ) sum = sum + *s++;
    return sum % 256;
    }
```

  - `hash(  "AB", 2 )` **and** `hash( "BA", 2 )` **return the same value!**
  - **This is called a collision**
  - **A variety of techniques are used for resolving collisions**

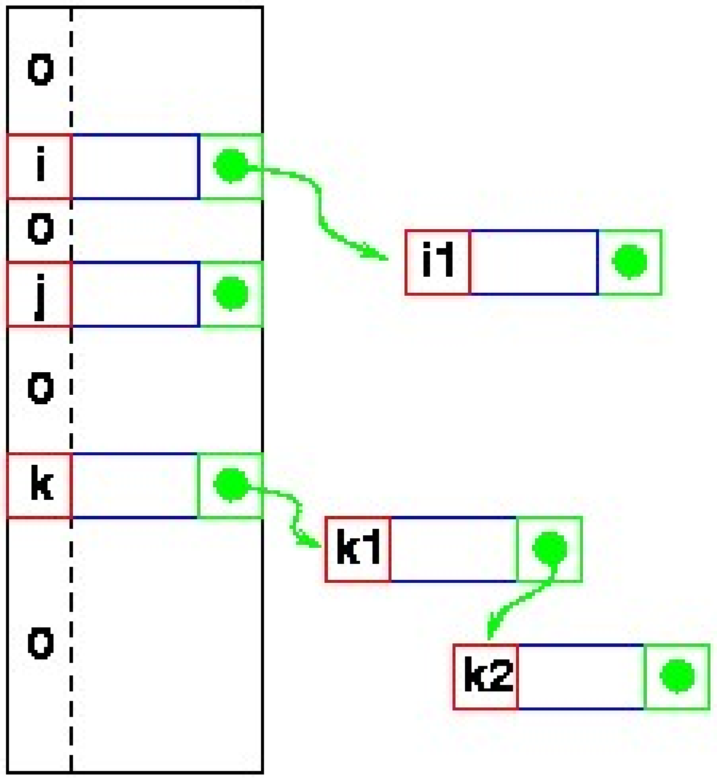
Hash Tables - Collision handling

- **Collisions**
  - **Occur when the hash function maps two different keys to the same address**
  - **The table must be able to recognise and resolve this**
  - **Recognise**
    - **Store the actual key with the item in the hash table**
    - **Compute the address**
      - $k = h(\text{key})$
    - **Check for a hit**
      - *if (* table[k].key == key *) then hit*
        else try next entry
  - **Resolution**
    - **Variety of techniques**

We'll look at various "*try next entry*" schemes

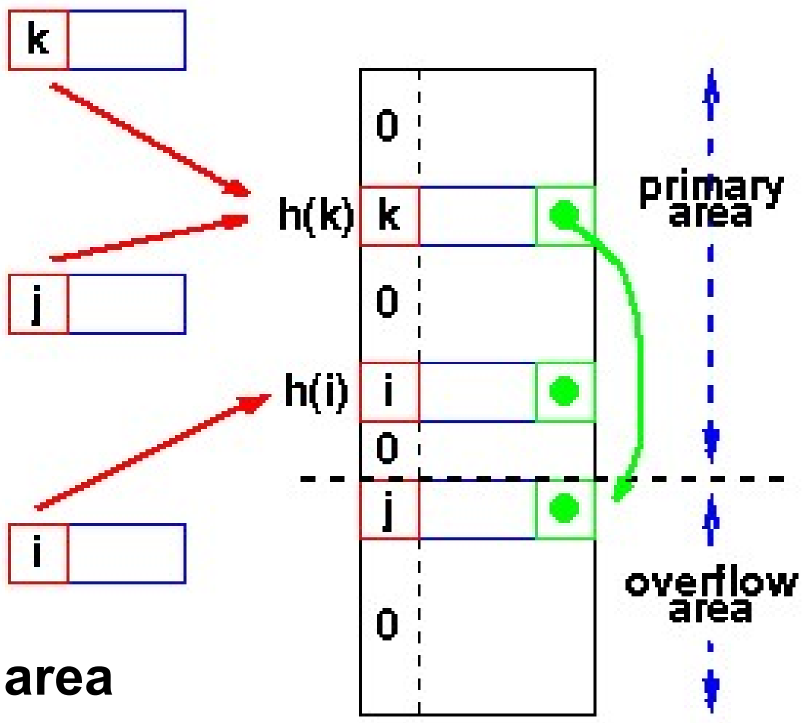
Hash Tables - Linked lists

- **Collisions - Resolution**
  - Linked list attached to each primary table slot
    - h(i) == h(i1)
    - h(k) == h(k1) == h(k2)
  - Searching for i1
    - Calculate h(i1)
    - Item in table, i, doesn't match
    - Follow linked list to i1
  - If `NULL` found, key isn't in table



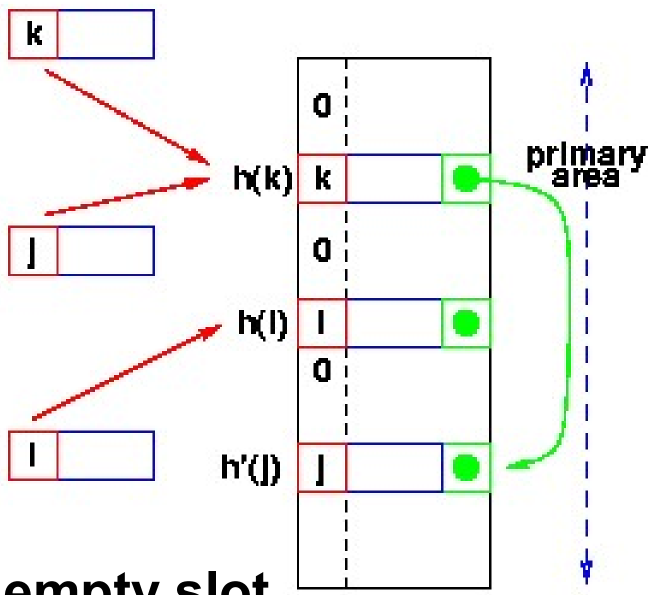
Hash Tables - Overflow area

- Overflow area
  - Linked list constructed in special area of table called overflow area
- h(k) == h(j)
- k stored first
- Adding j
  - Calculate h(j)
  - Find k
  - Get first slot in overflow area
  - Put j in it
  - k's pointer points to this slot
- Searching - same as linked list



Hash Tables - Re-hashing

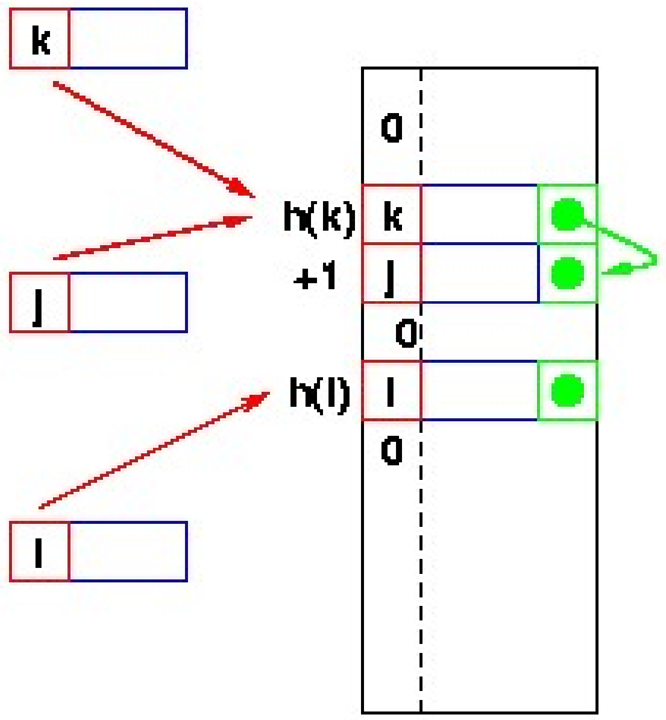
- Use a second hash function
  - Many variations
  - General term: re-hashing
- h(k) == h(j)
- k stored first
- Adding j
  - Calculate h(j)
  - Find k
  - Repeat for i until we find an empty slot
    - Calculate h'(j, i)
  - Put j in it
- Searching - Use h(x), then h'(x)



h'(x) -
second hash function

Hash Tables - Re-hash functions

- The re-hash function
  - Many variations
- Linear probing
  - h’(x) is +1
  - Go to the next slot until you find one empty



- Can lead to bad clustering
- Re-hash keys fill in gaps between other keys and exacerbate the collision problem

Hash Tables - Re-hash functions

- The re-hash function
  - Many variations
- Quadratic probing
  - $h'(x)$ is $c\,i^2$ on the i^{th} probe
  - Avoids primary clustering
  - Secondary clustering occurs
    - All keys which collide on $h(x)$ follow the same sequence
    - First
      - $a = h(j) = h(k)$
    - Then $a + c$, $a + 4c$, $a + 9c$, ...
    - Secondary clustering generally less of a problem

Hash Tables - Collision Resolution Summary

- **Chaining**
  - + **Unlimited number of elements**
  - + **Unlimited number of collisions**
  - - **Overhead of multiple linked lists**
- **Re-hashing**
  - + **Fast re-hashing**
  - + **Fast access through use of main table space**
  - - **Maximum number of elements must be known**
  - - **Multiple collisions become probable**
- **Overflow area**
  - + **Fast access**
  - + **Collisions don't use primary table space**
  - - **Two parameters (size/overflow size) which govern performance need to be estimated**

Hash Tables - Summary so far ...

- Potential $O(1)$ search time
  - If a suitable function $h(key) \rightarrow integer$ can be found
- Space for speed trade-off
  - "Full" hash tables don't work (more later!)
- Collisions
  - Inevitable
    - Hash function reduces amount of information in key
  - Various resolution strategies
    - Linked lists
    - Overflow areas
    - Re-hash functions
      - Linear probing h' is $+1$
      - Quadratic probing h' is $+ci^2$
      - Any other hash function!
        - or even sequence of functions!

Hash Tables - Choosing the Hash Function

- **"Almost any function will do"**
  - **But some functions are definitely better than others!**
- **Key criterion**
  - **Minimum number of collisions**
    - **Keeps chains short**
    - **Maintains $O(1)$ average**

Hash Tables - Choosing the Hash Function

- **Uniform hashing**
  - **Ideal hash function**
    - $P(k)$ = probability that a key, k, occurs
    - If there are m slots in our hash table,
    - a uniform hashing function, $h(k)$, would ensure:

$$\sum_{k \mid h(k) = 0} P(k) = \sum_{k \mid h(k) = 1} P(k) = \;\ldots\; \sum_{k \mid h(k) = m-1} P(k) = \frac{1}{m}$$

Read as sum over all k such that $h(k) = 0$

- *or, in plain English,*
- the number of keys that map to each slot is equal

Hash Tables - *A Uniform Hash Function*

- *If* the keys are integers randomly distributed in $[\,0\,,r\,)$, *then*

  Read as $0 \le k < r$

$$h(k) = \left\lfloor \frac{mk}{r} \right\rfloor$$

  is a uniform hash function
- Most hashing functions can be made to map the keys to $[\,0\,,r\,)$ for some r
  - *eg* adding the ASCII codes for characters $\mathrm{mod}\ 255$ will give values in $[\,0, 256\,)$ or $[\,0, 255\,]$
  - Replace + by xor

     same range without the mod operation

Hash Tables - Reducing the range to $[\,0, m\,)$

- **We've mapped the keys to a range of integers**

  $0 \le k < r$

- **Now we must reduce this range to $[\,0, m\,)$ where m is a reasonable size for the hash table**

- **Strategies**
  - **Division - use a mod function**
  - **Multiplication**
  - **Universal hashing**

Hash Tables - *Reducing the range to* $[\,0, m\,)$

 Division

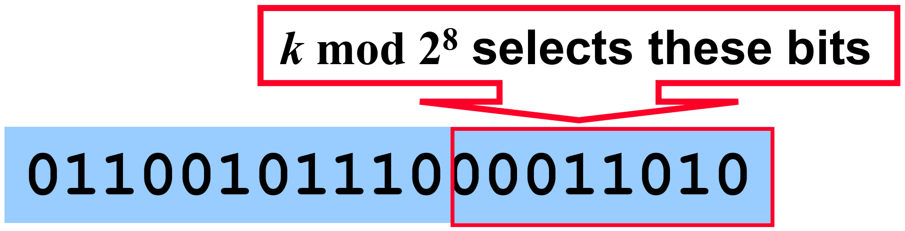
- Use a mod function

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

- Choice of m?
  - Powers of 2 are generally not good!

    $h(k) = k \bmod 2^n$

    selects last n bits of k



  - All combinations are not generally equally likely
- Prime numbers close to 2^n seem to be good choices

  eg want ~4000 entry table, choose $m = 4093$

Hash Tables - Reducing the range to $[\,0, m\,)$

- Multiplication method
  - Multiply the key by constant, A, $0 < A < 1$
  - Extract the fractional part of the product

    $(\,kA - \lfloor kA \rfloor)$
  - Multiply this by m

    $$h(k) = \lfloor m * (\,kA - \lfloor kA \rfloor) \rfloor$$
  - Now m is not critical and a power of 2 can be chosen
  - So this procedure is fast on a typical digital computer
    - Set $m = 2^p$
    - Multiply k (w bits) by $\lfloor A \cdot 2^w \rfloor$ ç $2w$ bit product
    - Extract p most significant bits of lower half
    - $A = ½(\sqrt{5} - 1)$ seems to be a good choice (*see* Knuth)

Hash Tables - Reducing the range to $(0, m]$

- Universal Hashing
  - A determined "adversary" can always find a set of data that will defeat any hash function
    - Hash all keys to same slot ç $O(n)$ search
  - Select the hash function randomly (at run time) from a set of hash functions
  - ---------
  - Functions are selected *at run time*
    - Each run can give different results
    - *Even with the same data*
    - Good average performance obtainable

Collision Frequency

- **Birthdays** *or* **the von Mises paradox**
  - There are 365 days in a normal year
    - Birthdays on the same day unlikely?
  - How many people do I need before "it's an even bet" (*ie* the probability is > 50%) that two have the same birthday?
  - View
    - the days of the year as the slots in a hash table
    - the "birthday function" as mapping people to slots
  - Answering von Mises' question answers the question about the probability of collisions in a hash table

Distinct Birthdays

- Let $Q(n)$ = probability that n people have distinct birthdays
- $Q(1) = 1$
- With two people, the 2nd has only 364 “free” birthdays

$$Q(2) = Q(1) * \frac{364}{365}$$

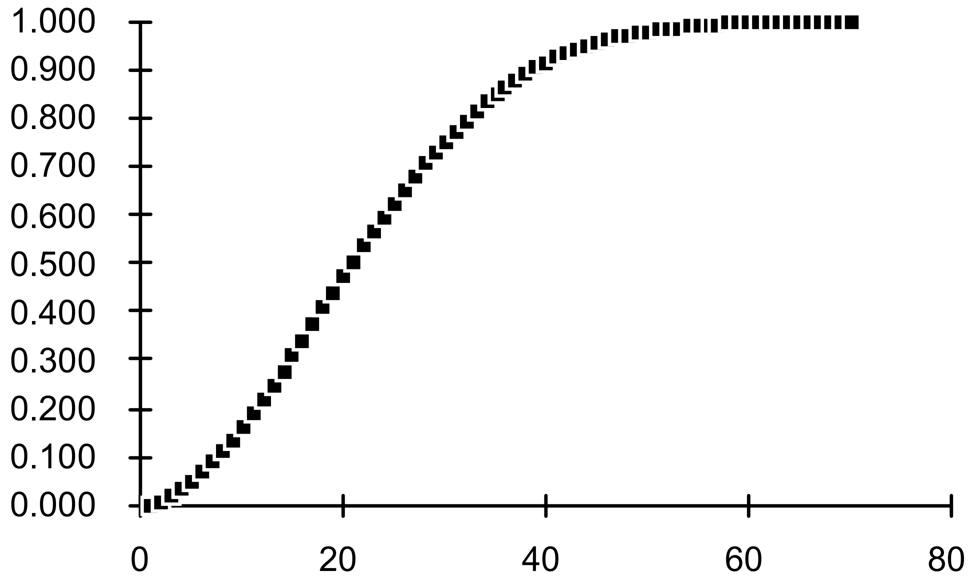
- The 3rd has only 363, and so on:

$$Q(n) = Q(1) * \frac{364}{365} * \frac{364}{365} * \ldots * \frac{365-n+1}{365}$$

Coincident Birthdays

- **Probability of having two identical birthdays**
- $P(n) = 1 - Q(n)$
- $P(23) = 0.507$

- **With 23 entries, table is only 23/365 = 6.3% full!**



Hash Tables - Load factor

- Collisions are very probable!
- Table load factor

$$\alpha = \frac{n}{m}$$

n = number of items

m = number of slots

  must be kept low
- Detailed analyses of the average chain length (or number of comparisons/search) are available
- Separate chaining
  - linked lists attached to each slot

gives best performance

  - but uses more space!
- Resizing can be done to increase the size of the hash table

Hash Tables - General Design

- Choose the table size
  - Large tables reduce the probability of collisions!
  - Table size, m
  - n items
  - Collision probability $\alpha = n / m$
- Choose a table organisation
  - Does the collection keep growing?
    - Linked lists (....... but consider a tree!)
  - Size relatively static?
    - Overflow area *or*
    - Re-hash
- Choose a hash function ....

Hash Tables - General Design

- Choose a hash function
  - A simple (and fast) one may well be fine ...
  - Read your text for some ideas!
- Check the hash function against your data
  - Fixed data
    - Try various h, m
      until the maximum collision chain is acceptable
    - Known performance
  - Changing data
    - Choose some representative data
    - Try various h, m until collision chain is OK
    - Usually predictable performance

Hash Tables - Review

- ***If you can meet the constraints***

+ **Hash Tables will generally give good performance**

+ $O(1)$ **search**