Data Structures and Algorithms (13)
(CS F222 / IS F222)

Hashing & YAHOO!

Udi Manber says that the three most important algorithms at **Yahoo** are

1. **hashing**
2. **hashing**
3. **hashing.**

Hashing has a variety of clever applications beyond just speeding up search, by giving you a short but distinctive representation of a larger document.

- ✔Is this new document different from the rest in a large corpus? – Hash the new document, and compare it to the hash codes of corpus.

- ✔ How can I convince you that a file isn't changed? – Check if the cryptographic hash code of the file you give me today is the same as that of the original. Any changes to the file will change the hash code.

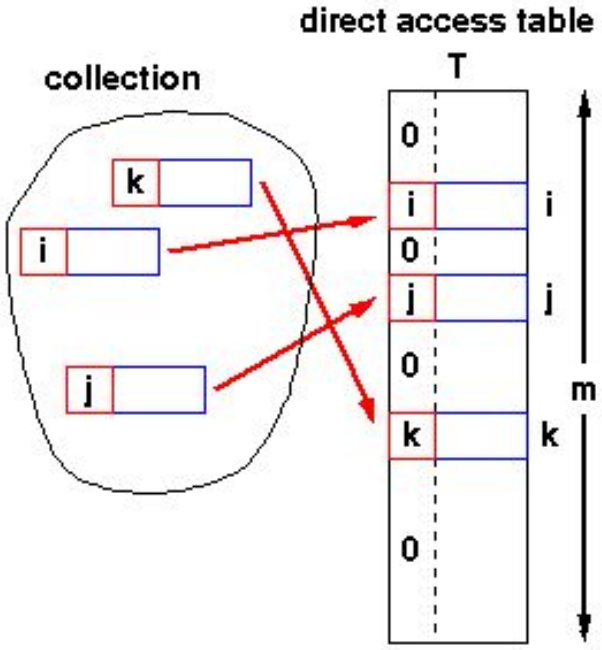
Hash Tables

- **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

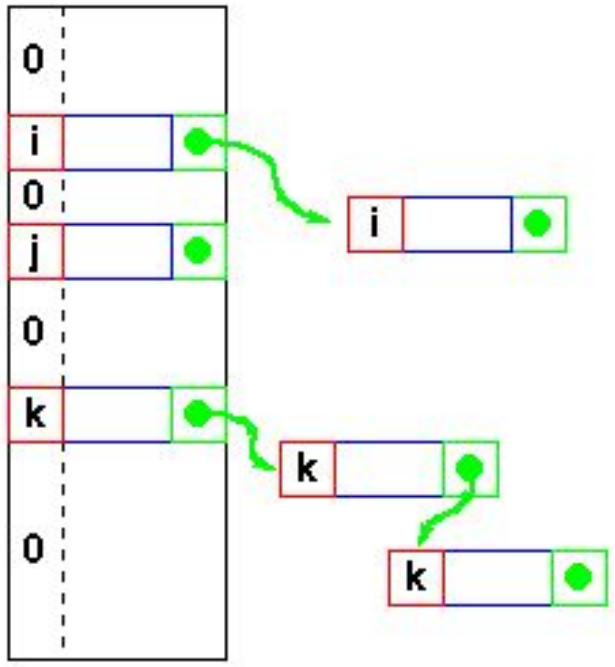
- **Constraints**
  - **Keys must be unique**
  - **Keys must lie in a small range**
  - **For storage efficiency, keys must be dense in the range**
  - **If they're sparse (lots of gaps between values), a lot of space is used to obtain speed**
    - **Space for speed trade-off**

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

    $h(\ key\ ) \quad \rightarrow \quad integer$

    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 Functions

- **Two parts:**
  - **Convert key into an integer in case the key is not an integer.**
    - **Done by the method hashCode().**
  - **Map an integer into a home bucket.**
    - **f(k) is an integer in the range [0, m-1], where m is the number of buckets in the table.**

- **A hash function is usually specified as the composition of two functions:**

  Hash code h_1**: keys → integers**

  Compression function h_2**: integers → [0,** N **- 1]**

- **The hash code is applied first, and the compression function is applied next on the result, i.e.,**
  $h(x) = h_2(h_1(x))$
- **The goal of the hash function is to "disperse" the keys in a random way**

Hash Tables - Hash functions

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

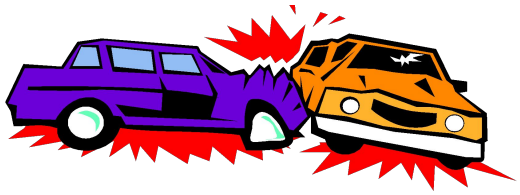
```
int hash(char *s, int n ) {
    int sum = 0;
    while( n-- ) sum = sum + *s++; // hashCode
    return sum % 256; // Compression Function
    }
```

    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

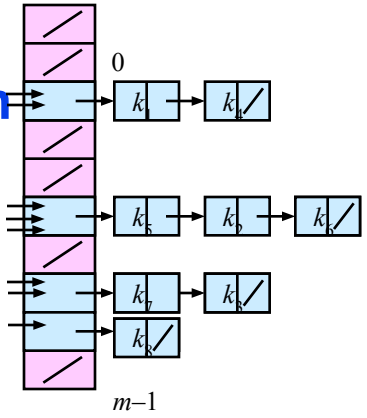
Hash Tables - Collision handling

- **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(key)
    - Check for a hit
      - *if (* table[k].key == key *) then hit else try next entry*
  - Resolution
    - Variety of techniques

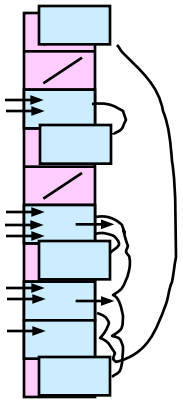
We'll look at various "*try next entry*" schemes

Methods of Resolution

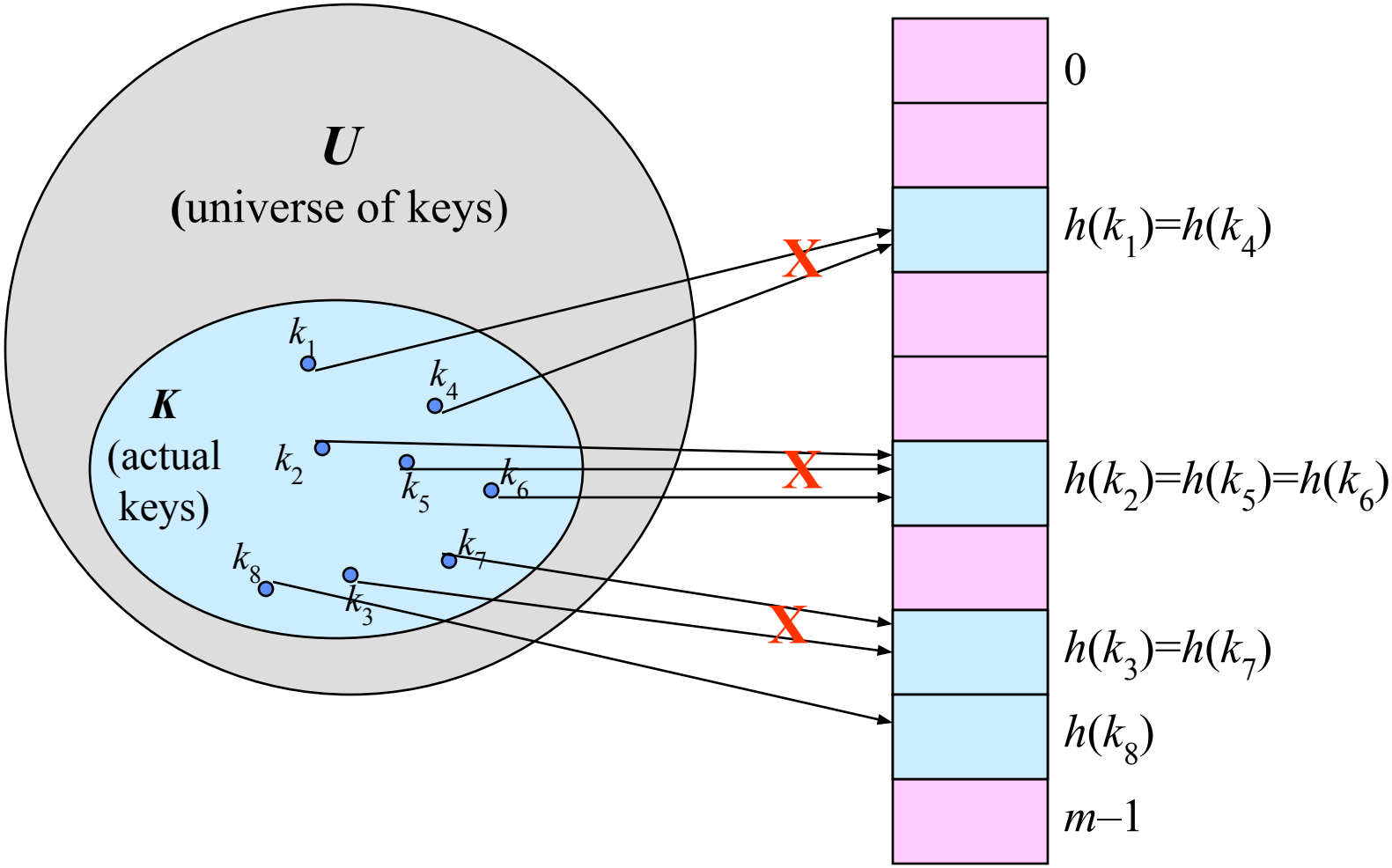
- **Chaining:**
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.



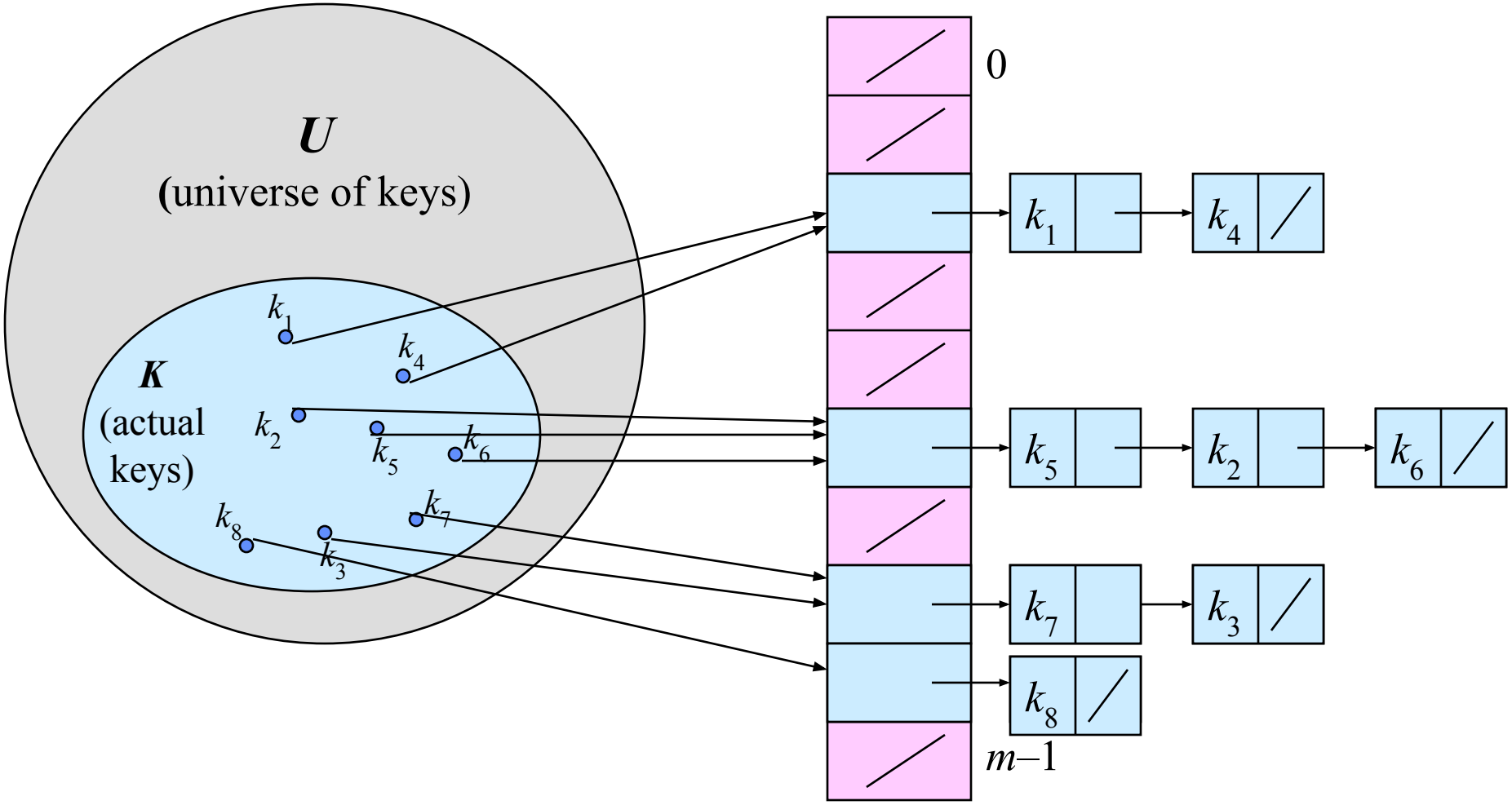
- **Open Addressing:**
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store elements in free slots of the table.



Collision Resolution by Chaining

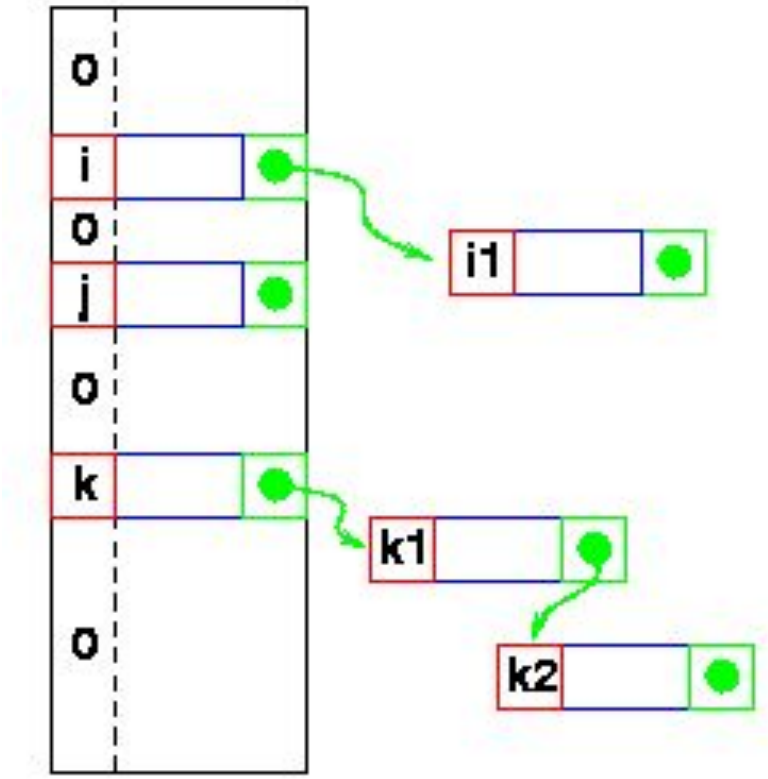


Collision Resolution by Chaining



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**



Theorem:
A successful search takes expected time $\Theta(1+\alpha)$.

Load factor $\alpha = n/m$ = average keys per slot.

m – number of slots.

n – number of elements stored in the hash table.

Hashing with Chaining

Dictionary Operations:

- **Chained-Hash-Insert (*T, x*)**
  - **Insert *x* at the head of list *T*[*h*(*key*[*x*])].**
  - **Worst-case complexity – *O*(1).**

- **Chained-Hash-Delete (*T, x*)**
  - **Delete *x* from the list *T*[*h*(*key*[*x*])].**
  - **Worst-case complexity – proportional to length of list with singly-linked lists. *O*(1) with doubly-linked lists.**

- **Chained-Hash-Search (*T, k*)**
  - **Search an element with key *k* in list *T*[*h*(*k*)].**
  - **Worst-case complexity – proportional to length of list.**

Analysis on Chained-Hash-Search

- Load factor $\alpha = n/m$ = average keys per slot.
  - m – number of slots.
  - n – number of elements stored in the hash table.

- Worst-case complexity: $\Theta(n)$ + time to compute $h(k)$.

- Average depends on how h distributes keys among m slots.

- Assume
  - *Simple uniform hashing*.
    - Any key is equally likely to hash into any of the m slots, independent of where any other key hashes to.
  - $O(1)$ time to compute $h(k)$.

- Time to search for an element with key k is $\Theta(|T[h(k)]|)$.

- Expected length of a linked list = load factor = $\alpha = n/m$.

Expected Cost of an Unsuccessful Search

Theorem:
An unsuccessful search takes expected time $\Theta(1+\alpha)$.

Proof:

- ✔ Any key not already in the table is equally likely to hash to any of the m slots.
- ✔ To search unsuccessfully for any key k, need to search to the end of the list $T[h(k)]$, whose expected length is α.
- ✔ Adding the time to compute the hash function, the total time required is $\Theta(1+\alpha)$.

Expected Cost of a Successful Search

Theorem:
A successful search takes expected time $\Theta(1+\alpha)$.

Proof:

- The probability that a list is searched is proportional to the number of elements it contains.
- Assume that the element being searched for is equally likely to be any of the n elements in the table.
- The number of elements examined during a successful search for an element x is 1 more than the number of elements that appear before x in x's list.
  - These are the elements inserted *after* x was inserted.
- Goal:
  - Find the average, over the n elements x in the table, of how many elements were inserted into x's list after x was inserted.

Expected Cost of a Successful Search

Theorem:
A successful search takes expected time $\Theta(1+\alpha)$.

Proof (contd):

- Let x_i be the i^{th} element inserted into the table, and let $k_i = key[x_i]$.
- Define indicator random variables $X_{ij} = I\{h(k_i) = h(k_j)\}$, for all i, j.
- Simple uniform hashing $\Rightarrow \Pr\{h(k_i) = h(k_j)\} = 1/m$
  $\Rightarrow E[X_{ij}] = 1/m$.
- Expected number of elements examined in a successful search is:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

No. of elements inserted after x_i into the same slot as x_i.

Proof – Contd.

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right]$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{ij}]\right) \quad \text{(linearity of expectation)}$$

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)$$

$$=1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)$$

$$=1+\frac{n-1}{2m}$$

$$=1+\frac{\alpha}{2}-\frac{\alpha}{2n}$$

Expected total time for a successful search = Time to compute hash function + Time to search

$= O(2+\alpha/2 - \alpha/2n) = O(1+\alpha)$.

Expected Cost – Interpretation

- If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$.

  ⇒ Searching takes constant time on average.

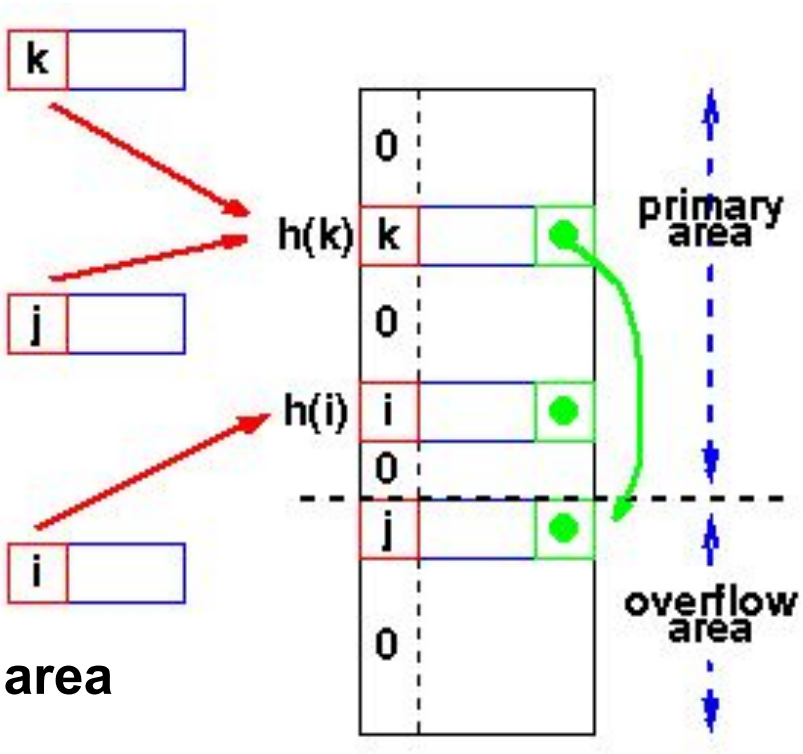
- Insertion is $O(1)$ in the worst case.

- Deletion takes $O(1)$ worst-case time when lists are doubly linked.

- Hence, all dictionary operations take $O(1)$ time on average with hash tables with chaining.

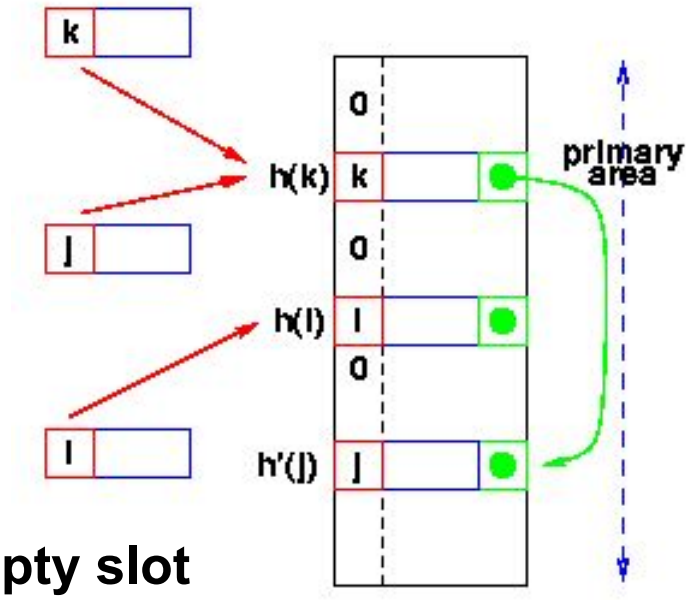
Hash Tables - Overflow area

- 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

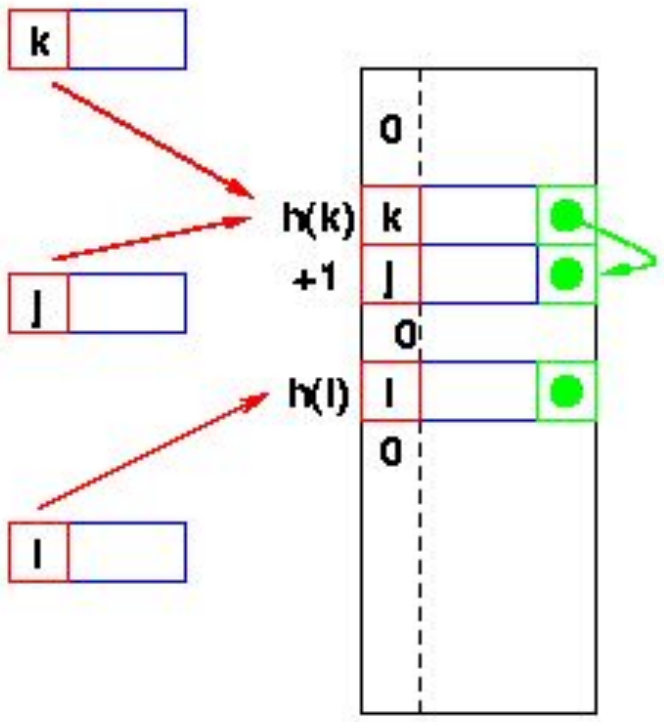
- Use a second hash function
  - Many variations
  - General term: re-hashing
- h(k) == h(j)
- k stored first
- Adding j
  - Calculate h(j)
  - Find k
  - Repeat until we find an empty slot
    - Calculate h'(j)
  - Put j in it
- Searching - Use h(x), then h'(x)



h'(x) - second hash function

Hash Tables - Re-hash functions

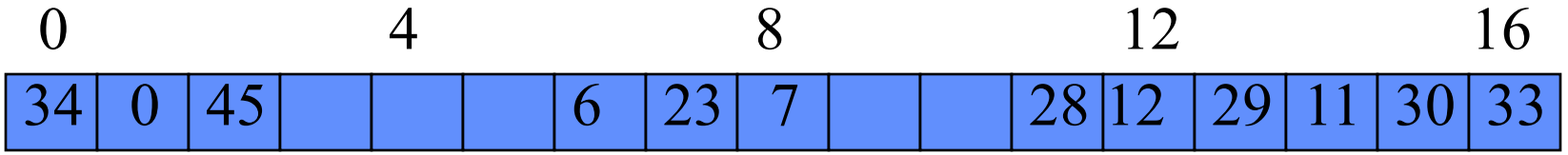
- The re-hash function
  - Many variations
- Linear probing
  - h'(x) is +1
  - Go to the next slot until you find one empty



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

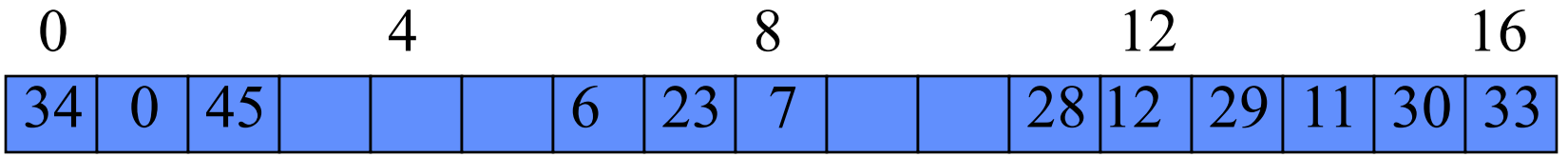
Linear Probing – Get And Put

- **divisor = b (number of buckets) = 17.**
- **Home bucket = key % 17.**

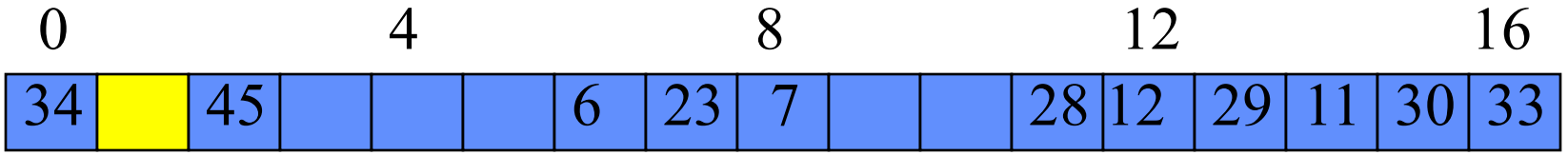


- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45

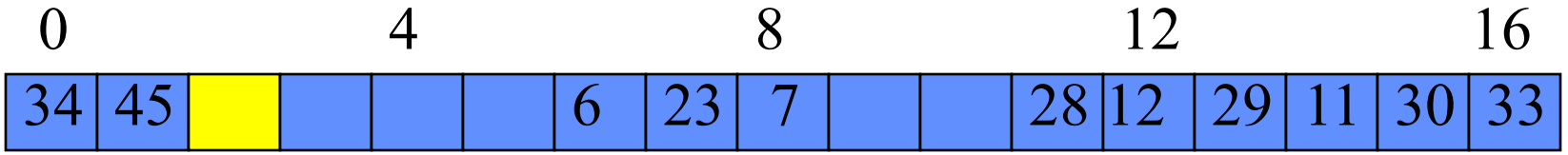
Linear Probing – Remove



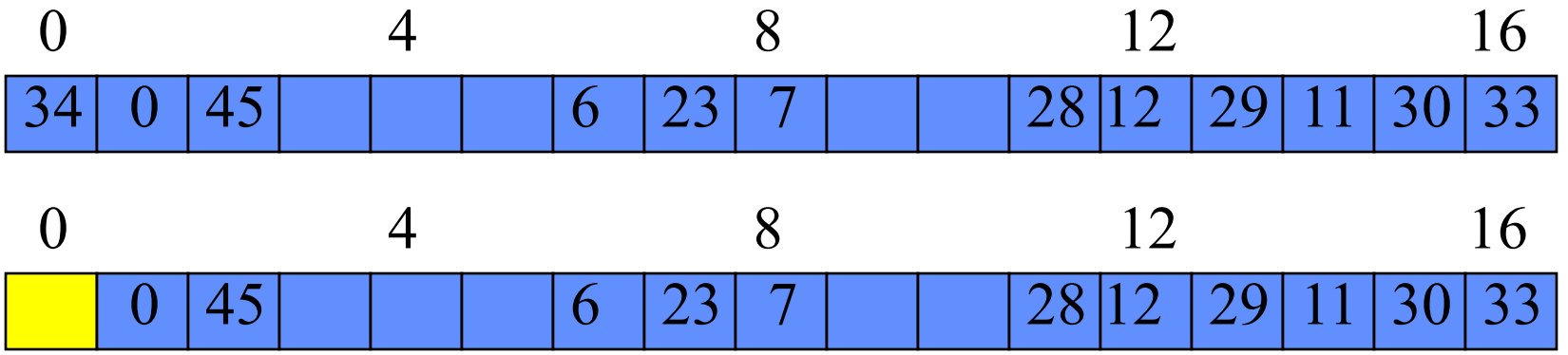
- **remove(0)**



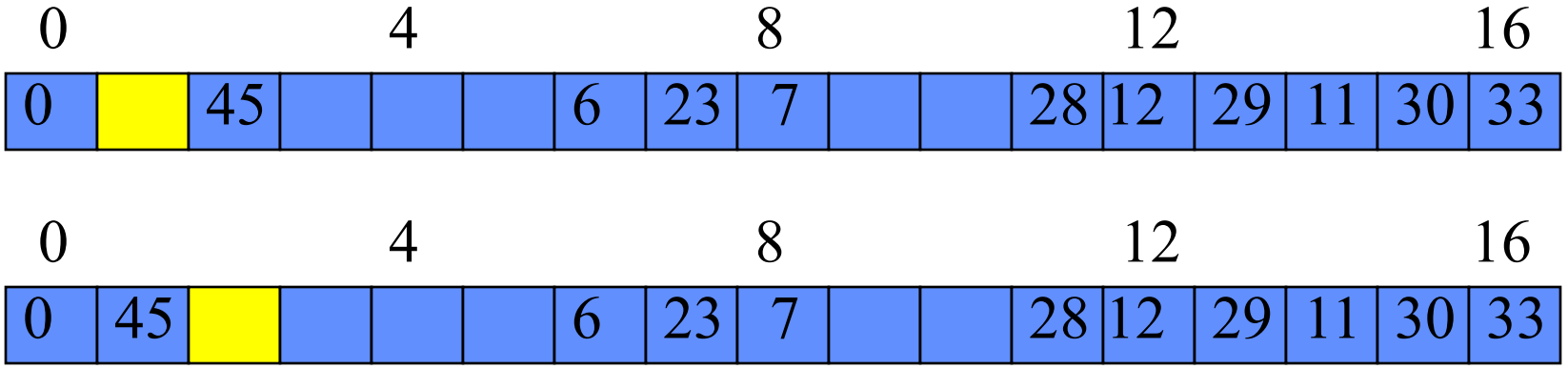
- Search cluster for pair (if any) to fill vacated bucket.



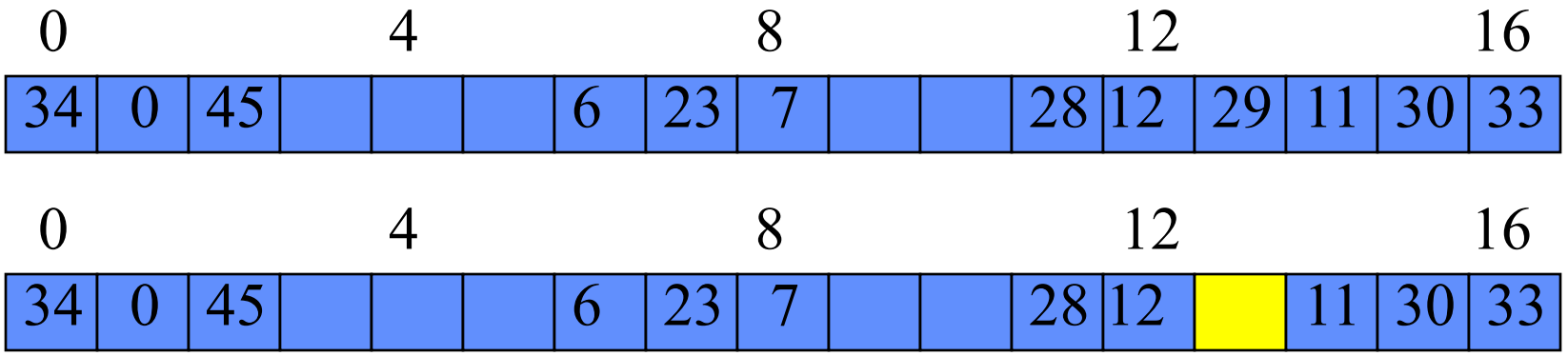
Linear Probing – remove(34)



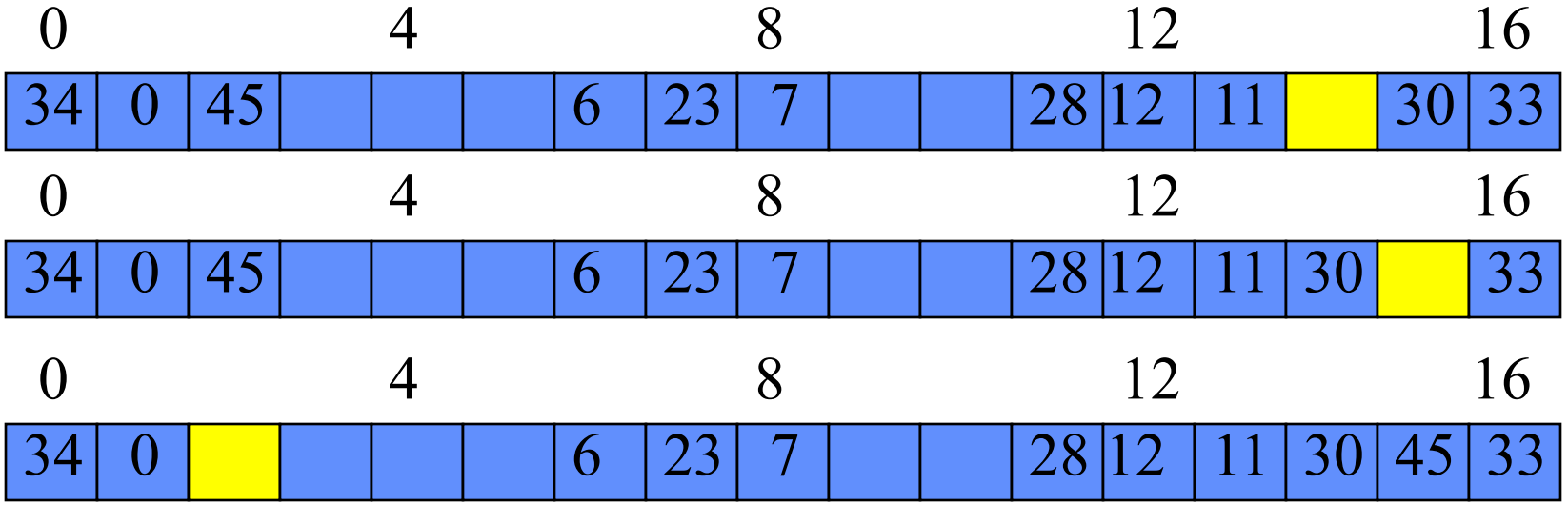
- **Search cluster for pair (if any) to fill vacated bucket.**



Linear Probing – remove(29)



- **Search cluster for pair (if any) to fill vacated bucket.**



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 $a + c,\ a + 4c,\ a + 9c, \ldots$
    - Secondary clustering generally less of a problem

Theorem:
The expected number of probes in an unsuccessful search in an open-address hash table is at most $1/(1-\alpha)$.

Hash Tables - Summary so far ...

- **Potential *O(1)* search time**
  - **If a suitable function *h(key)* → *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*

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

  is a uniform hash function

  Read as $0 \le k < r$

- Most hashing functions can be made to map the keys to $[\,0\,,\,r\,)$ for some r
  - *eg* adding the ASCII codes for characters 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** $\bmod$ **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!**
    - **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 - 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

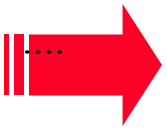
- **Detailed analyses of the average chain length (or number of comparisons/search) are available**
- **Separate chaining**
  - **linked lists attached to each slot**

gives best performance

  - **but uses more space!**

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

  - ***Like radix sort*,**
    they rely on *calculating* an address from a key
  - **But, *unlike radix sort*,**
    relatively easy to get good performance
    - ***with a little experimentation***
      - ***∴ not advisable for unknown data***
    - ***collection size relatively static***
    - ***memory management is actually simpler***
      - **All memory is pre-allocated!**

Linear Probing

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

key Probe number Auxiliary hash function

Quadratic Probing

- $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \quad c_1 \neq c_2$

key Probe number Auxiliary hash function

Double Hashing

- $h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$