Design and Analysis of Algorithms

Lecture 02: Hashing

Ahmed Hamdy

Goal

- Perform operations (Search, Insert and Delete) as fast as possible

Data structure	Search	Insert	Delete (given pos.)
Array (unsorted)	$O(n)$	$O(1)$	$O(n)$
Array (sorted)	$O(\log n)$	$O(n)$	$O(n)$
Linked List (unsorted)	$O(n)$	$O(1)$	$O(1)$
Linked List (sorted)	$O(n)$	$O(n)$	$O(1)$
Binary Search Trees (BST)	$O(\log n)$ /$O(n)$	$O(\log n)/O(n)$	$O(\log n)/O(n)$

- Can we do better?
- Hash tables: $O(1)$

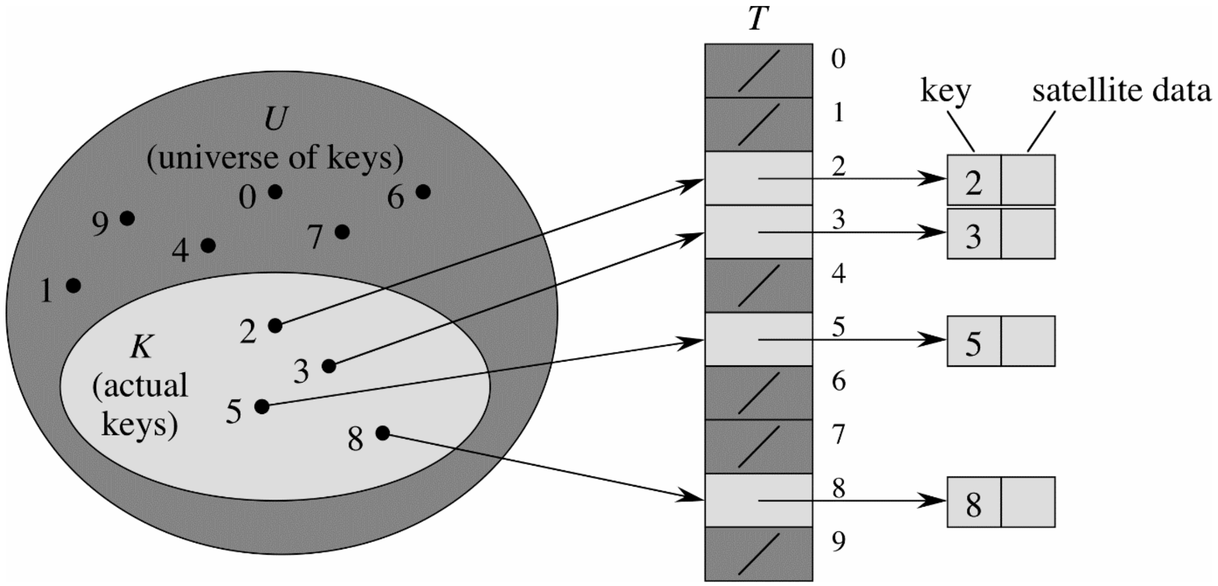
Direct Address Table

DIRECT-ADDRESS-SEARCH(T, k)
  return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)
  $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)
  $T[x.key] = \text{NIL}$

Each of these operations takes only $O(1)$ time.



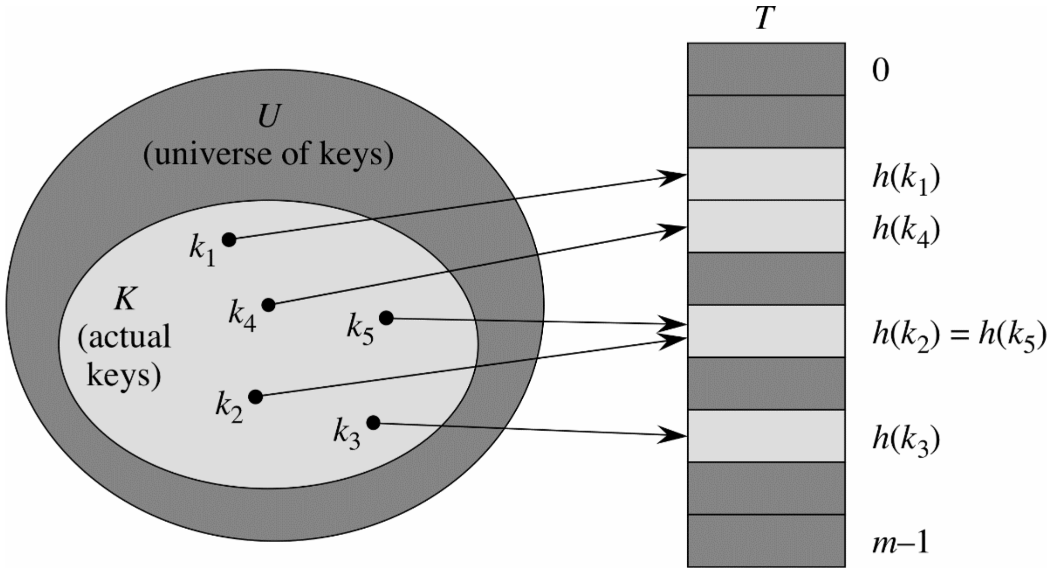
$$U = \{0, 1, \ldots, m-1\}$$

- Operations take $O(1)$ time.
- Data can be stored in table itself without a linked-list.
- Disadv.: If universe U is large.
- NOT a hash table.

Hash Table

- Reduce table size to m
- Use hash function

$$h : U \to \{0, 1, \ldots, m-1\}$$



- Problem: collision
- Resolution:
  - Chaining
  - Open addressing

Hash Function

- Goal: minimize collisions

- Keys must be integers
  - Convert non-integers to integers, i.e. strings

- For strings, use the ASCII of each character to build an integer.
- Example, ‘pt’ → (112,116) → $112 * 128 + 116 = 14452$

Division Method

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

- m should be prime not too close to powers of 2
- Example:
  - $n = 2000$
  - $\alpha = 3$
  - $m = 701$
  - $h(k) = k \ \ mod \ \ 701$

Multiplication Method

$$h(k) = \lfloor m(k\,A\,mod\;1) \rfloor, \qquad 0 < A < 1$$

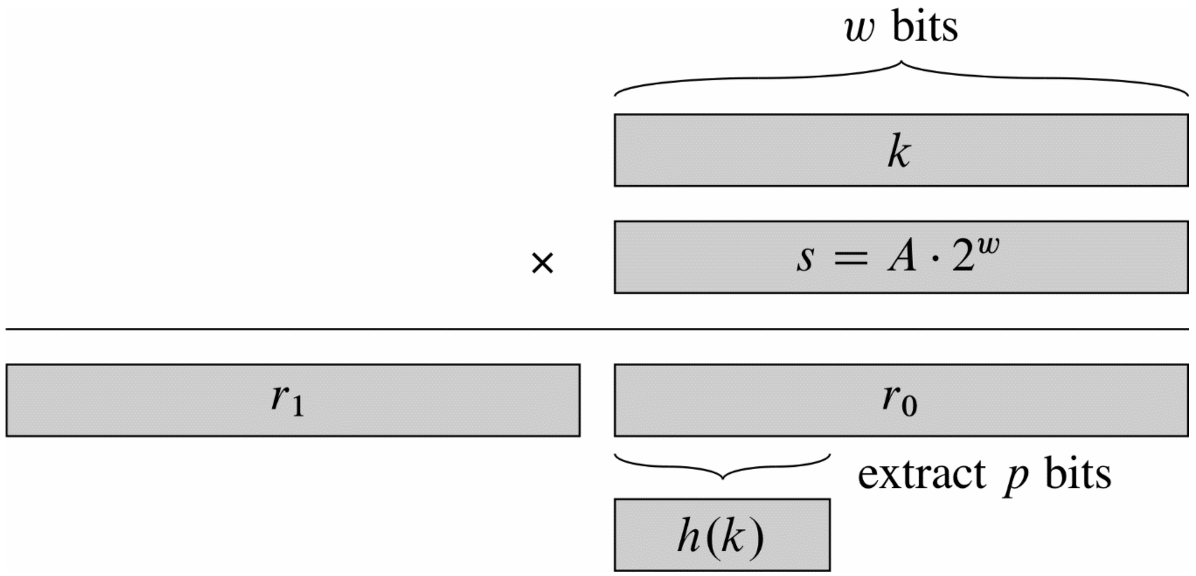
- Extracts the fractional part of $k\ A$ and then multiplies by m (hash table size)
- m better be 2^P
- Assume w, the word size in machine (i.e. 32 bits)
- k fits one word
- Restrict A to $s/2^w$

Multiplication Method

$$h(k) = \lfloor m(k\,A\,mod\,1) \rfloor, \qquad 0 < A < 1$$

Optimization

- Assume w, the word size (i.e. 32 bits)
- k fits one word
- Restrict A to $s/2^w$
- Compute:



$$ks = r_1 2^w + r_0 \xrightarrow{mod} r_0 \xrightarrow{\times m} q_1 2^p + q_0 \xrightarrow{floor} q_1$$

integer **fraction** **integer** **fraction** **shift right w-p bits**

Multiplication Method

$$h(k) = \lfloor m(k\,A \bmod 1) \rfloor, \qquad 0 < A < 1$$

- $k = (0\ 0\ 1\ 1)_b = 3$
- $A = 0.875 = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} = (0.1\ 1\ 1\ 0)_b$
- $s = A \times 2^4 = 2^4\left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8}\right) = 2^3 + 2^2 + 2^1 = (1\ 1\ 1\ 0)_b = 14$
- $k \times s = 3 \times 14 = 42 = (0\ 0\ 1\ 0\quad 1\ 0\ 1\ 0)_b = (r_1\ r_0)_b$
- $k \times A = 3 \times 0.875 = 2\frac{5}{8} = 2.625 = k \times \frac{s}{2^4} = \frac{42}{2^4} = (0\ 0\ 1\ 0\ .\ \mathbf{1}\ \mathbf{0}\ 1\ 0)_b$
- $k \times A \bmod 1 = 0.r_0 = \frac{5}{8} = \frac{1}{2} + \frac{1}{8} = (.\ \mathbf{1}\ \mathbf{0}\ 1\ 0)_b$
- $2^p \times 0.r_0 = 2^2 \times \frac{5}{8} = \frac{5}{2} = (\mathbf{1}\ \mathbf{0}\ .\ 1\ 0)_b$ (assuming $p = 2$)
- $h(k) = \lfloor 2^p \times 0.r_0 \rfloor = (\mathbf{1}\ \mathbf{0})_b$

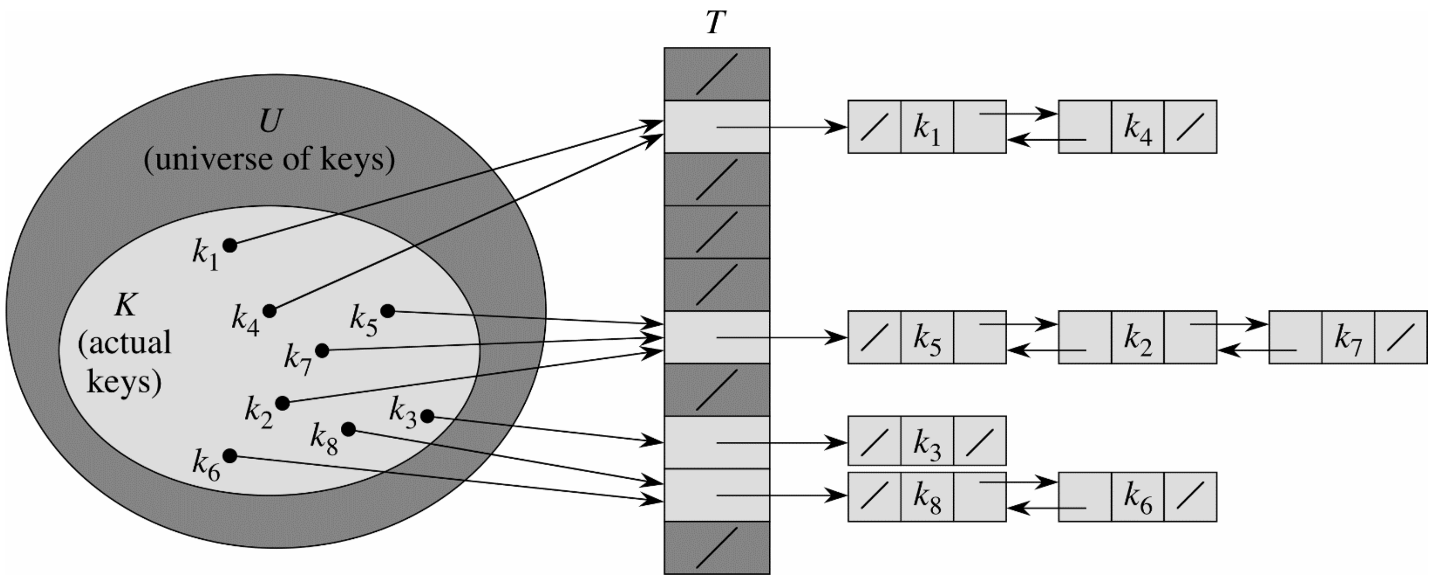
$$\begin{array}{rcc} & & \boxed{0\,0\,1\,1} \\ \times & & \boxed{1\,1\,1\,0} \\ \hline & \boxed{0\,0\,1\,0} & \boxed{\mathbf{1}\,\mathbf{0}\,1\,0} \end{array}$$

Universal hashing

- An adversary can choose the data to *all hash* to the same position if he determines the hashing function
- The problem happens because of determinism!!
- Solution: to avoid determinism, use *randomization*
- At every execution over the *whole dataset*, choose the hash function randomly from a set of hash functions.
- This hash function remains the same during the *whole run*, otherwise does not make sense.

Chaining

- Collisions are resolved by storing in linked lists
- Elements in the main array are pointers to the linked lists
- Load factor $\alpha = \frac{n}{m}$



- Unsuccessful search takes $\Theta(1 + \alpha)$
- Can be $\Theta(1)$ if $\alpha = O(1)$, in other words $n = O(m)$

Open Addressing

- Elements occupy hash table itself
- To resolve collisions, instead of inserting in a linked list, examine a *next empty* position to store the element
- Examining the *next* position is called *probing*

```
HASH-INSERT(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == NIL
        T[j] = k
        return j
    else i = i + 1
until i == m
error "hash table overflow"
```

i^{th} Next position

Open Addressing

Searching inside hash

- Must search all possible *next* positions

```
HASH-SEARCH(T, k)
i = 0
repeat
    j = h(k, i)
    if T[j] == k
        return j
    i = i + 1
until T[j] == NIL or i == m
return NIL
```

Open Addressing

Deletion is tricky

- If we delete an element while there are elements inserted after it, we can't search after deletion because the element is marked as nil
- Solution: mark with a different mark
  - Problem: search time no longer depends on α
- Open addressing is not typically used if keys can be deleted, use *chaining* instead

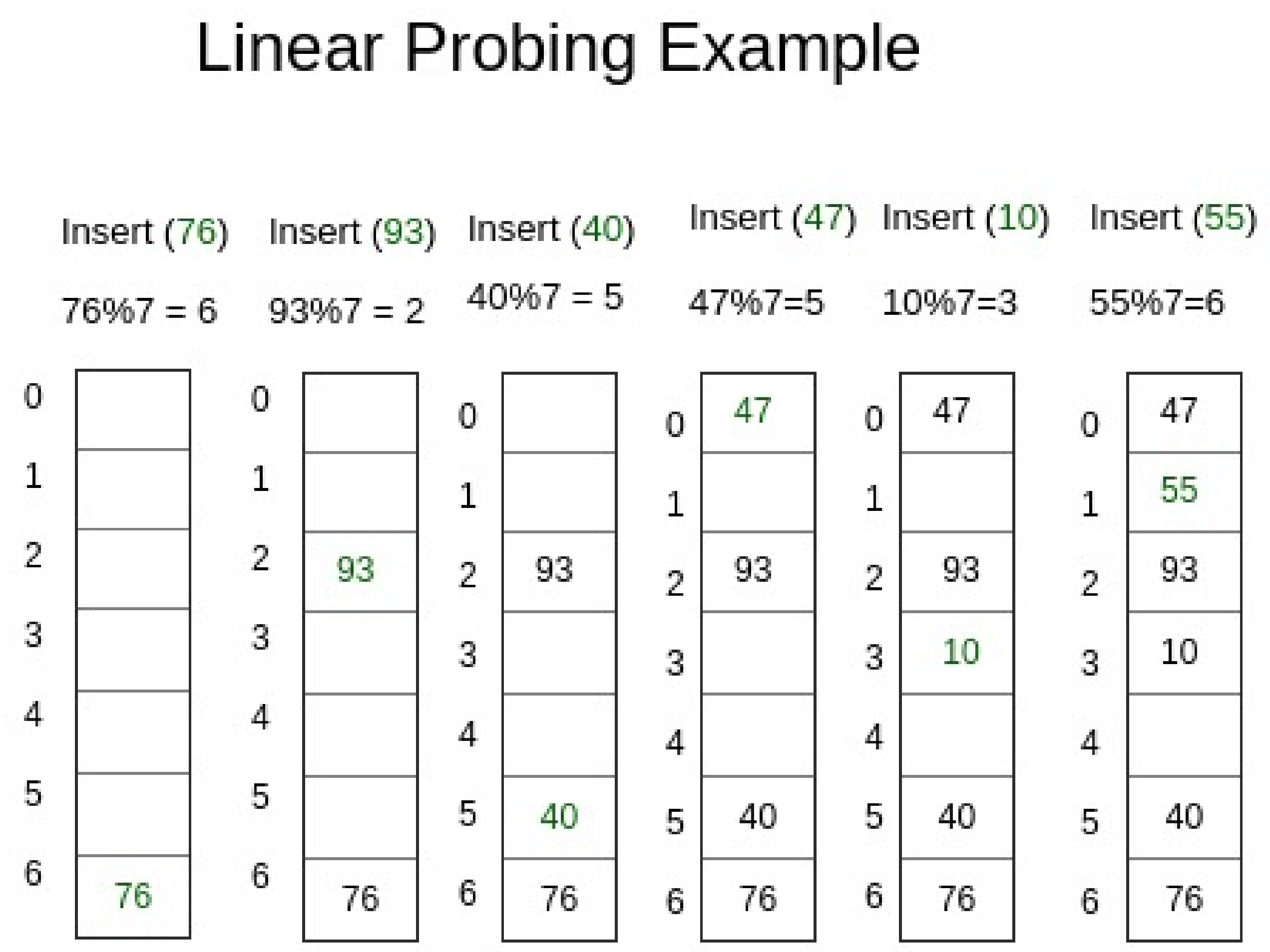
Open Addressing

Linear probing

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

- Examine consecutive positions $(mod\ m)$
- Problem: *primary clustering*
  - An empty slot preceded by i full slots will be filled with probability $(i + 1)/m$

Open Addressing



Courtesy of Geeksforgeeks.com

Open Addressing

Quadratic probing

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m \;, \tag{11.5}$$

where h' is an auxiliary hash function, c_1 and c_2 are positive auxiliary constants, for $i = 0, 1, \ldots, m-1$

- Jump by quadratic steps
- Works much better than linear probing
- Problem: if $h(k_1, 0) = h(k_2, 0)$, then $h(k_1, i) = h(k_2, i)$, called *secondary clustering*
  - Still initial position determines the entire sequence

Open Addressing

Double hashing

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

where h_1 and h_2 are auxiliary hash functions

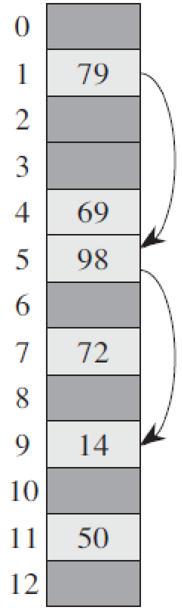


Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, we insert the key 14 into empty slot 9, after examining slots 1 and 5 and finding them to be occupied.

Open Addressing

- Analysis

Theorem 11.6
Given an open-address hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

- When:
  - $\alpha = 0.5, \frac{1}{1-\alpha} = 2$
  - $\alpha = 0.9, \frac{1}{1-\alpha} = 10$
  - $\alpha = 0.99, \frac{1}{1-\alpha} = 100$

Perfect Hashing

- Use two levels of hashing

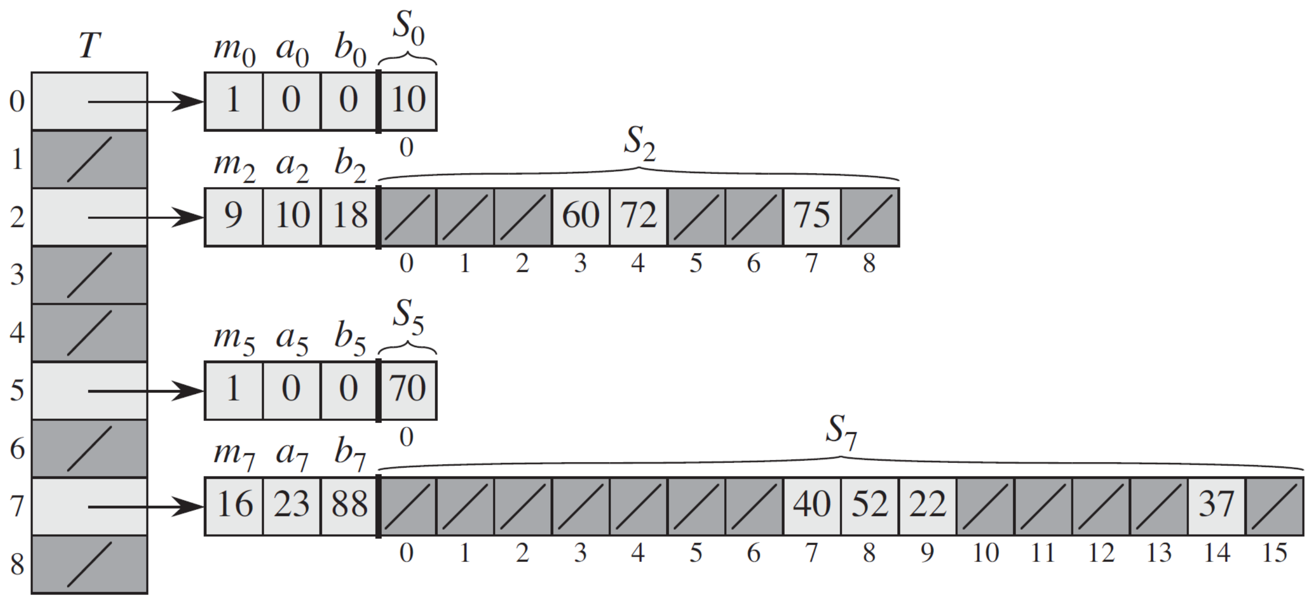


Figure 11.6 Using perfect hashing to store the set $K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, and so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is $m_j = n_j^2$, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 7$, key 75 is stored in slot 7 of secondary hash table S_2. No collisions occur in any of the secondary hash tables, and so searching takes constant time in the worst case.

Perfect Hashing

- Good choice for *static* data
  - Examples:
    - Reserved words in programming language
    - Set of file names in CD-ROM
- $O(1)$ memory access in the worst case
- Universal hashing is used at each level
- Similar to chaining but use a *secondary hash table* instead of *linked lists*
- Careful selection of the secondary hash table can avoid collisions