Chapter11. Hash Tables

- Hash table
- Issue with hashing
- Collision Resolution Techniques
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing

Review

- Array Lists
  - O(1) access
  - O(n) insertion (average case)
  - O(n) deletion (average case)
- Linked Lists
  - O(n) access
  - O(n) insertion (average case)
  - O(n) deletion (average case)
- Binary Search Trees
  - O(log n) access if balanced
  - O(log n) insertion if balanced
  - O(log n) deletion if balanced

Review

- What is hashing? Why is it useful to us?
  - There are lots of applications that need to support only the operations INSERT, SEARCH, and DELETE.
    These are known as "dictionary" operations.
- Applications:
  - data base search
    - books in a library
    - patient records, GIS data etc.
  - web page caching (web search)
  - combinatorial search (game tree)

Review : Performance goal for dictionary operations:

- *O(n) is too inefficient*.

Goal

- O(log n) on average
- O(log n) in the worst-case
- O(1) on average

Data structure that achieve these goals:

O(log n) on average ⇨ binary search tree(BST)

O(log n) in the worst-case ⇨ balanced BST(AVL tree)

O(1) on average ⇨ hashing. (but worst-case is O(n))

Hash

hash: transitive verb[1]

1. (a) to chop (as meat and potatoes) into small pieces
   (b) confuse, muddle
2. ...

Hash brown

Review

- Hashing
  - important and widely useful technique for implementing dictionaries
  - Technique supporting insertion, deletion, and search in *average-case constant time: O(1)*
  - Operations requiring elements to be sorted (e.g. find minimum) are not efficiently supported

Dictionary & Hash Tables

- **Dictionary:**
  - Dynamic-set data structure for storing items indexed using *keys*.
  - Supports operations Insert, Search, and Delete.
  - Applications:
    - Symbol table of a compiler.
    - Memory-management tables in operating systems.
    - Large-scale distributed systems.
- **Hash Tables:**
  - Effective way of implementing dictionaries.
  - Generalization of ordinary arrays.

Direct-address Tables

- Direct-address Tables are ordinary arrays.
- Facilitate direct addressing.
  - Element whose key is k is obtained by indexing into the k^{th} position of the array.
- Applicable when we can afford to allocate an array with one position for every possible key.
  - i.e. when the universe of keys U is small.
- Dictionary operations can be implemented to take $O(1)$ time.

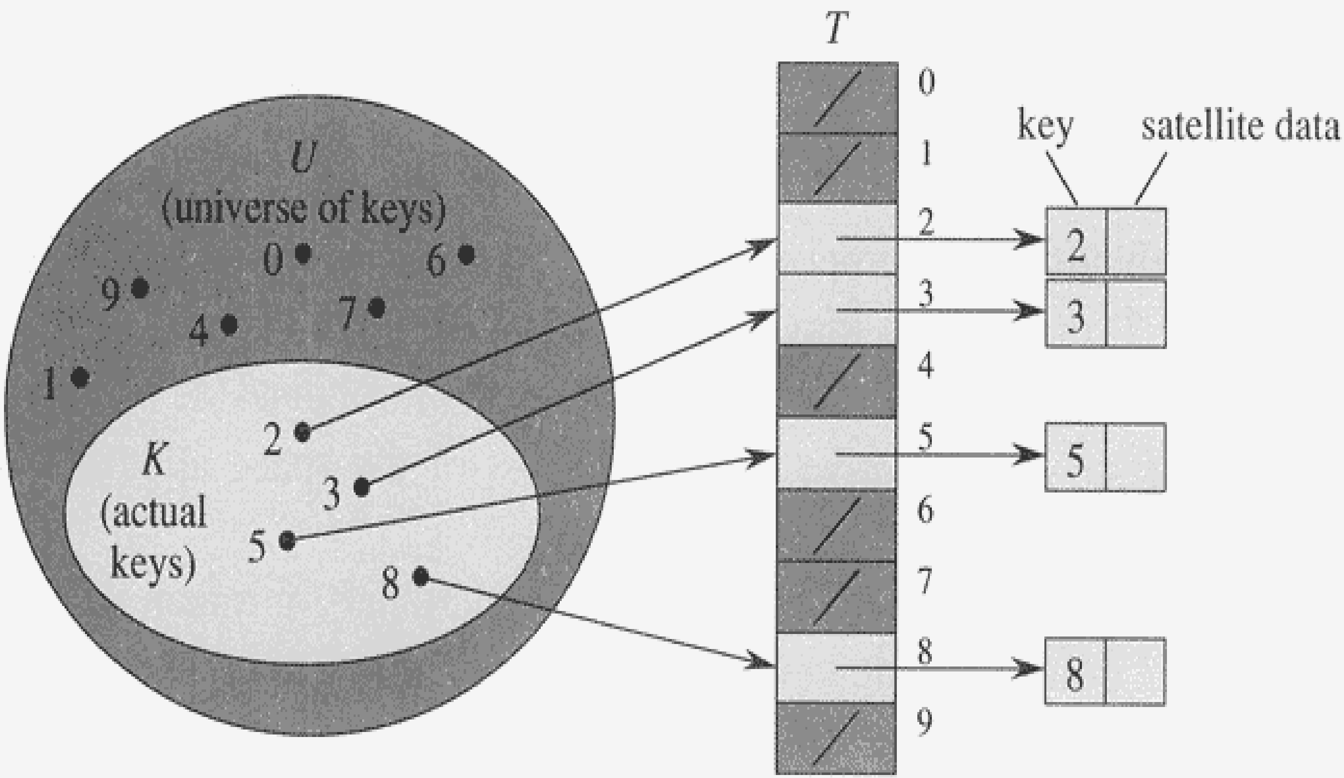
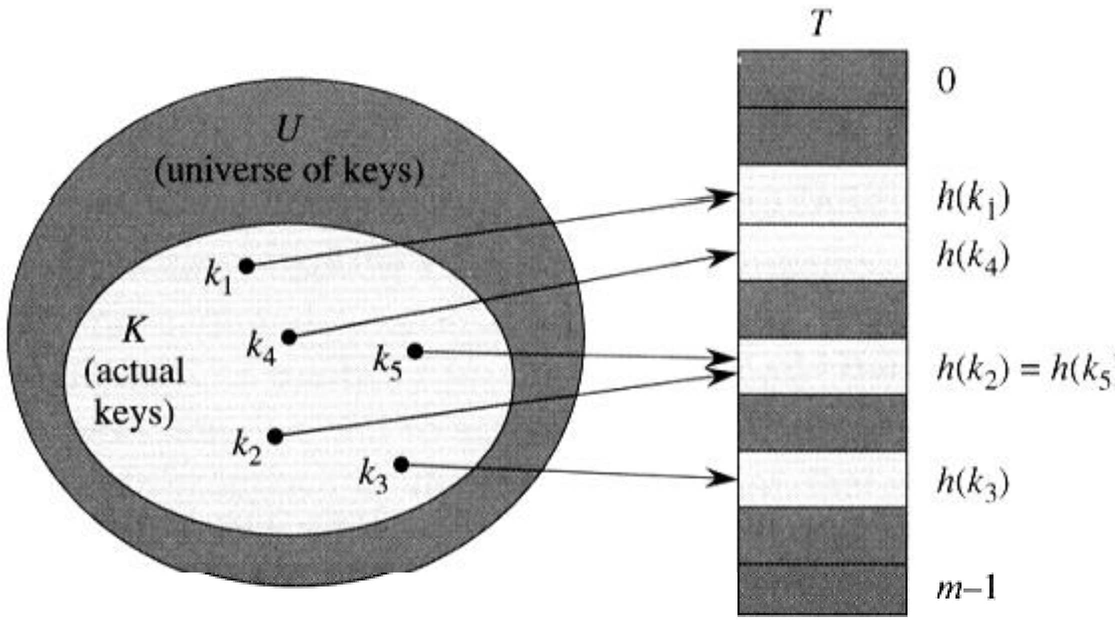


Figure 11.1 Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

Hash Table

- The difficulty with direct address is obvious: if the universe U is large, storing a table T of size $|U|$ may be impractical, or even impossible.
- Furthermore, the set K of keys actually stored may be so small relative to U. Specifically, the storage requirements can be reduced to $O(|K|)$, even though searching for an element in the hash table still requires only $O(1)$ time.



U
(universe of keys)
K
(actual
keys)
k_1
k_4
k_5
k_2
k_3
T
0
$h(k_1)$
$h(k_4)$
$h(k_2) = h(k_5)$
$h(k_3)$
$m-1$

Hash Table

- **Notation:**
  - U : Universe of all possible keys.
  - K : Set of keys actually stored in the dictionary.
  - $|K| = n$.
- When U is very large,
  - Arrays are not practical.
  - $|K| << |U|$.
- Use a table of size proportional to $|K|$: ***hash tables.***
  - However, we lose the direct-addressing ability.
  - Define functions that map keys to slots of the hash table.

Hash function

- Hash function h:
  Mapping from U to the slots of a hash table $T[0..m-1]$.

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

- With arrays, key k maps to slot $A[k]$.
- With hash tables, key k maps or "hashes" to slot $T[h[k]]$.
- $h[k]$ is the *hash value* of key k.

Hash function example

- elements = Integers
- *h(i) = i % 10 (= i mod 10)*
- add 41, 34, 7, and 18
- constant-time lookup:
  - just look at *i % 10* again later
- Hash tables have <u>no ordering information</u>!
  - Expensive to do following:
    - getMin, getMax, removeMin, removeMax,
    - the various ordered traversals
    - printing items in sorted order

0	
1	41
2	
3	
4	34
5	
6	
7	7
8	18
9	

Hash Functions

- A hash function transforms a key into a table address
- What makes a **good hash function**?
  (1) Easy to compute
  (2) Minimize the number of collisions
  (3) Unbiased
- Approximates a random function: for every input, every output is equally likely (simple uniform hashing)
- In practice, it is very hard to satisfy the simple uniform hashing property

Good Approaches for Hash Functions

- Minimize the chance that closely related keys hash to the same slot
  - Strings such as *pt* and *pts* should hash to different slots
- Derive a hash value that is independent from any patterns that may exist in the distribution of the keys

The Division Method

- **Idea:**
  - Map a key k into one of the m slots by taking the remainder of k divided by m

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

- **Advantage**:
  - fast, requires only one operation
- **Disadvantage**:
  - Certain values of *m* are bad, e.g.,
    - power of 2
    - non-prime numbers
- Good choice for *m*:
  - Primes, not too close to power of 2 (or 10) are good.

Example : The Division Method

- If $m = 2^p$, then h(k) is just the least significant p bits of k
  - p = 1 $\Rightarrow$ m = 2

    $\Rightarrow$ h(k) = {0,1} , least significant 1 bit of k
  - p = 2 $\Rightarrow$ m = 4

    $\Rightarrow$ h(k) ={0,1,2,3} , least significant 2 bits of k
- Choose m to be a prime, not close to a power of 2(or 10)
  - Column 2: k mod 97
  - Column 3: k mod 100

	m 97	m 100
16838	57	38
5758	35	58
10113	25	13
17515	55	15
31051	11	51
5627	1	27
23010	21	10
7419	47	19
16212	13	12
4086	12	86
2749	33	49
12767	60	67
9084	63	84
12060	32	60
32225	21	25
17543	83	43
25089	63	89
21183	37	83
25137	14	37
25566	55	66
26966	0	66
4978	31	78
20495	28	95
10311	29	11
11367	18	67

The Multiplication Method

Idea:

- Multiply key k by a constant A, where $0 < A < 1$
- Extract the fractional part of kA
- Multiply the fractional part by m
- Take the floor of the result

$$h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$$

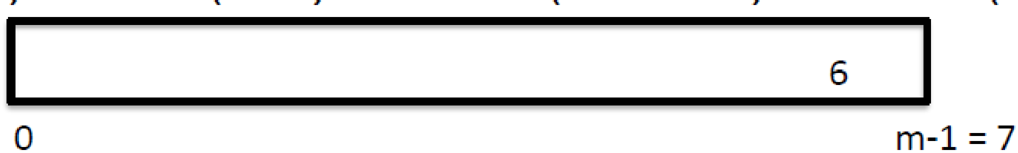
- **Disadvantage:** Slower than division method
- **Advantage:** Value of m is not critical, (e.g., typically 2^p)

Example : Multiplication Method

Example:

Key k = 3; m = 8 slots
(1) A = .61
(2) kA = 3 x .61 = 1.83

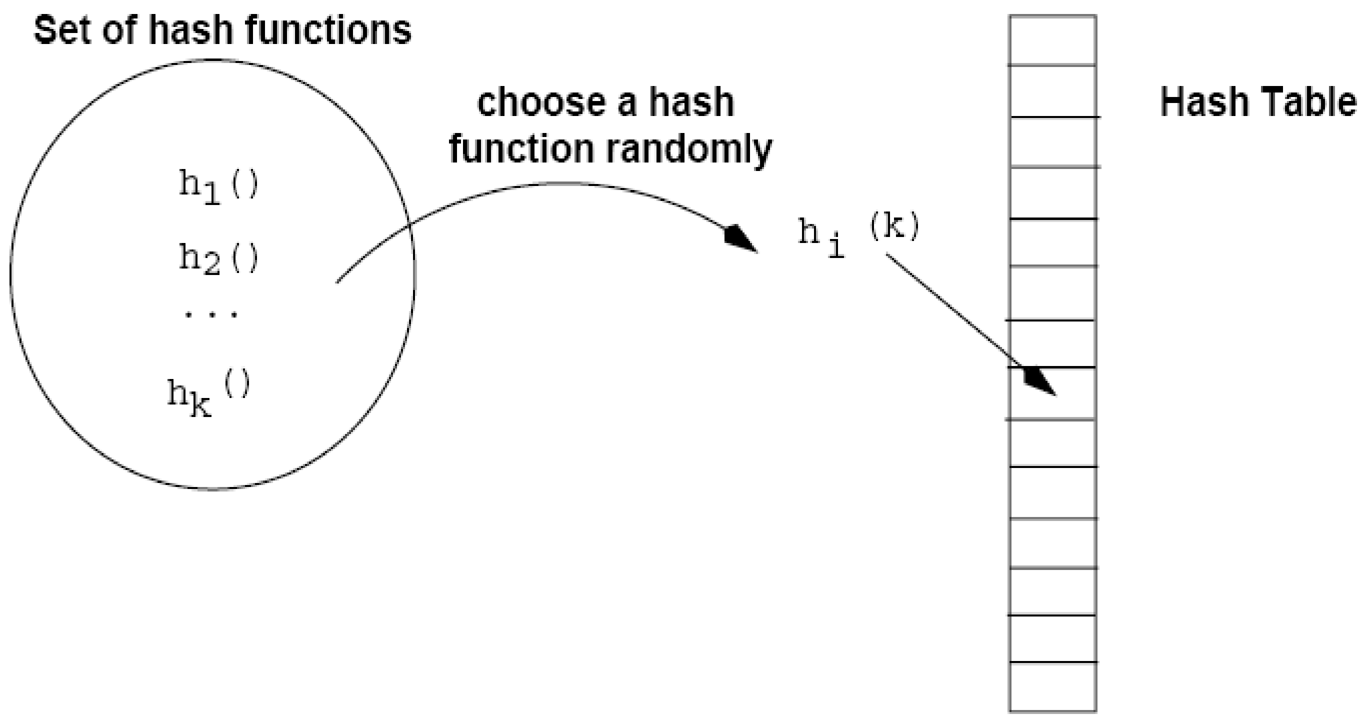
(3) Floor (f m) = Floor(.83 x 8) = Floor(6.64) = 6



Universal Hashing

- A malicious adversary who has learned the hash function chooses keys that all map to the same slot, giving worst-case behavior.
- Defeat the adversary using **Universal Hashing**
  - Use a different random hash function each time.
  - Ensure that the random hash function is independent of the keys that are actually going to be stored.
  - Ensure that the random hash function is "good" by carefully designing a class of functions to choose from.
    - Design a **universal** class of functions.

Universal Hashing



Definition of Universal Hash Functions

H=(*h*(*k*): U --h(k)--> (0,1,...,m-1))

H is said to be universal if

for $x \neq y$, **|(h() ∈ H: h(x)=h(y)|=|H|/m**

(notation: |H|: number of elements in H - cardinality of H)

- The chance of a collision between two keys is the 1/*m* chance of choosing two slots randomly & independently.
- Universal hash functions give good hashing behavior

Universal Hashing

- What is the probability of collision in this case ?

  It is equal to the probability of choosing a function $h \in U$ such that $x \neq y$ --> $h(x) = h(y)$ which is

  $$\Pr(h(x)=h(y)) = \frac{|H|/m}{|H|} = \frac{1}{m}$$

- With universal hashing the chance of collision between distinct keys k and l is no more than the 1/m chance of collision if locations h(k) and h(l) were randomly and independently chosen from the set {0, 1, …, m – 1}

Advantages of Universal Hashing

- Universal hashing provides good results on average, independently of the keys to be stored
- Guarantees that no input will always elicit the worst-case behavior
- Poor performance occurs only when the random choice returns an inefficient hash function
  (this has small probability)

Issue with Hashing

- Multiple keys can hash to the same slot
  - <u>Collisions</u>(two keys hash to same slot) are possible.
  - Design hash functions such that collisions are minimized.
  - But avoiding collisions is impossible.
    - Design collision-resolution techniques.
- Search will cost $\Theta(n)$ time in the worst case.
  - However, all operations can be made to have an expected complexity of $\Theta(1)$.

Collision

- Two or more keys hash to the same slot.
- For a given set K of keys
  - If $|K| \le m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions completely is hard, even with a good hash function