CSE 201: Data Structures

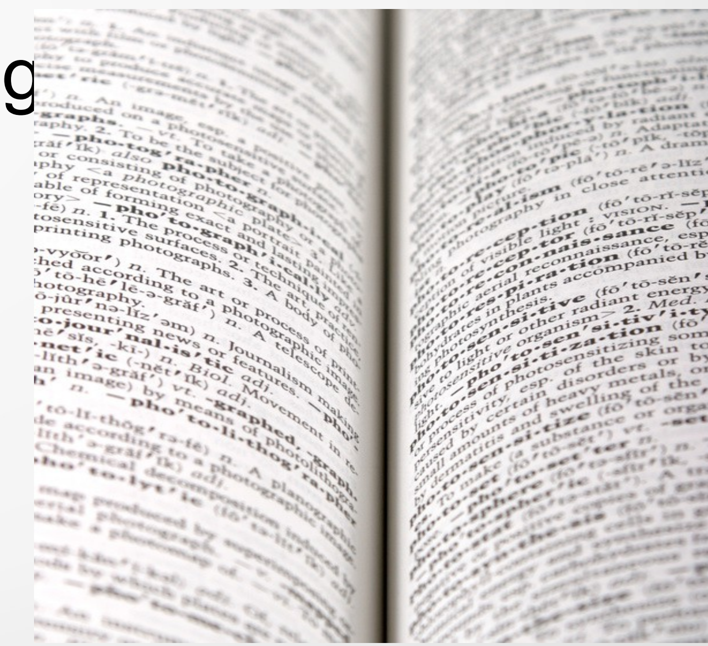
Lecture 11: Dictionaries and Hash Tables

Dr. Vidhya Balasubramanian

Amrita School of Engineering
Amrita Vishwa Vidyapeetham

Dictionaries

- Models a searchable collection of key-element items
  - Multiple items with same key allowed
- Main operations include
  - insertion, searching, and deleting
- Applications
  - Telephone directory
  - Mapping student info to roll nos



Dictionary ADT

- find(k): if the dictionary has an item with key k, returns the position of this item,

  else, returns a null position.
- insertItem(k, o): inserts item o with key k into the dictionary
- removeElement(k): removes the item with key k from the dictionary. Exception of no such element.
- Other functions
  - size(), isEmpty()
  - keys(), Elements()

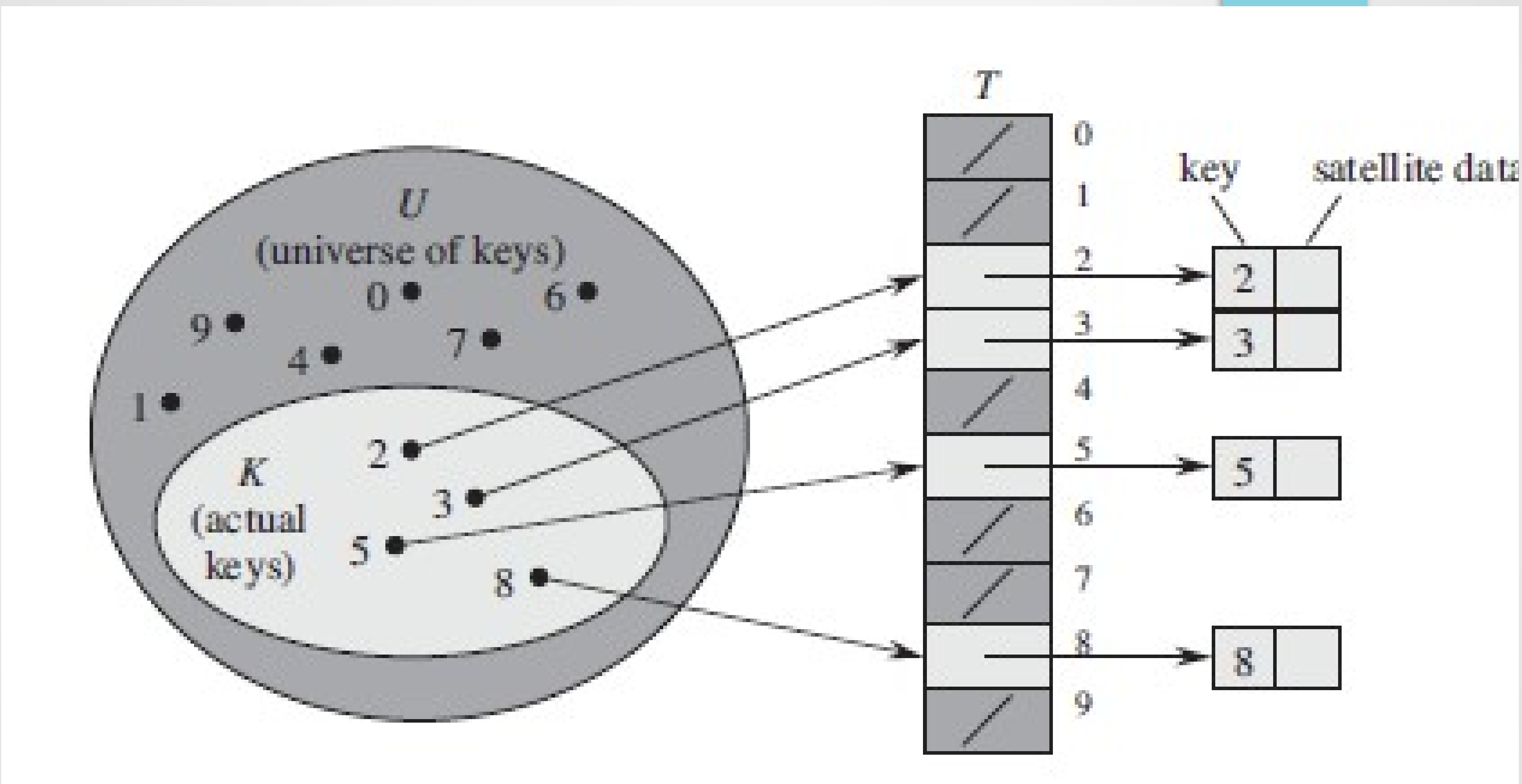
Dictionaries

- Types
  - Ordered Dictionaries
    - A total order relation is defined on the keys
  - Unordered Dictionaries
    - No order relation is assumed on the keys
    - Only equality testing between keys is used
- Associative Stores
  - When keys are unique, keys are like addresses to the location where the element is stored

Direct Addressing

- Applied when the number of keys are small and are unique
- Use an array, or direct-address table, denoted by T[0..m-1], in which each position, or slot, corresponds to a key in the universe U
  - Key k is stored in slot k
  - If the set contains no elements, then the slot is empty
- When the universe is large this is impractical
  - Use hashing

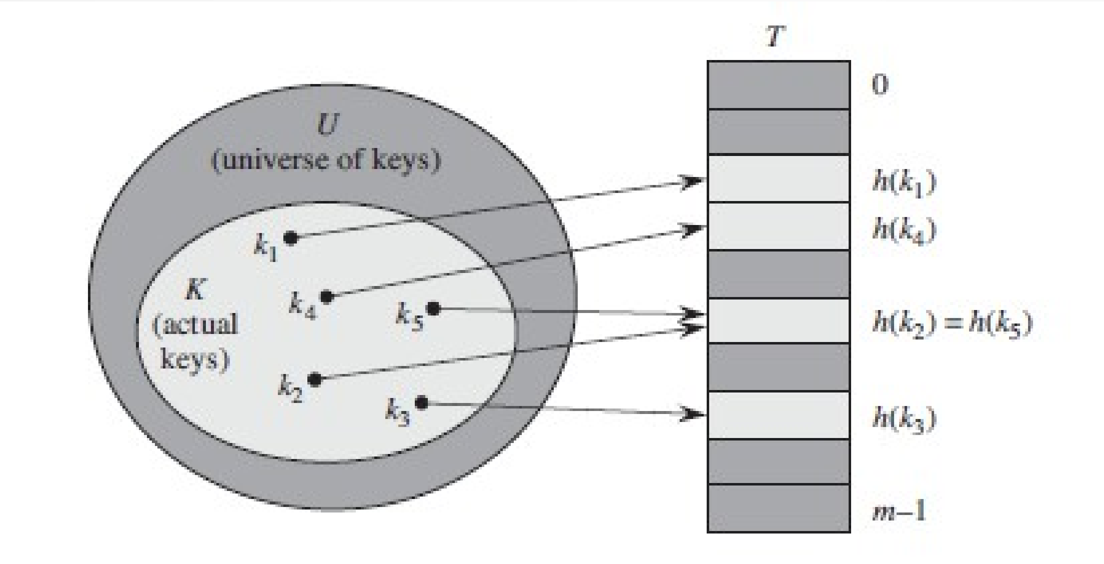
Direct Addressing



Src: CLRS Textbook

Hashing

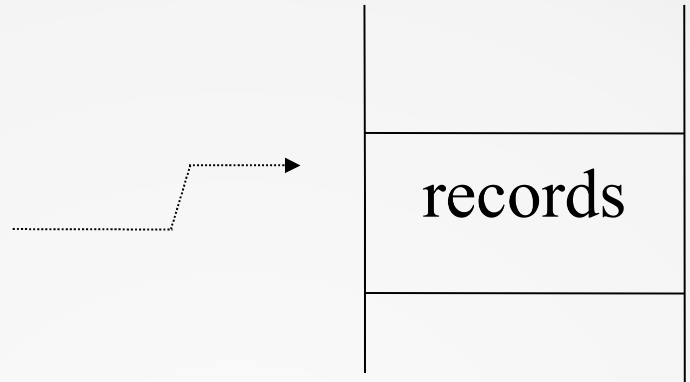
- Element k is stored in slot h(k)
  - Use a hash function to compute slot from the key
  - h maps the universe U of keys into the slots of a hash table T[0..m-1]
  - Size m is much lesser than the |U|



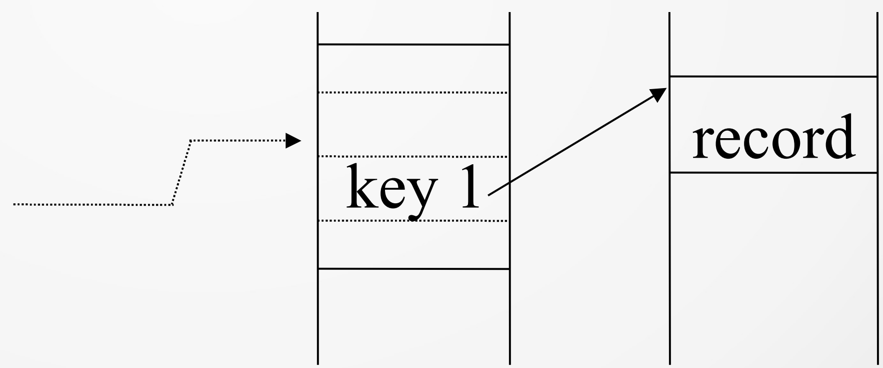
Src: CLRS Text

Alternatives

(1) key -> h(key)



(2) key -> h(key)



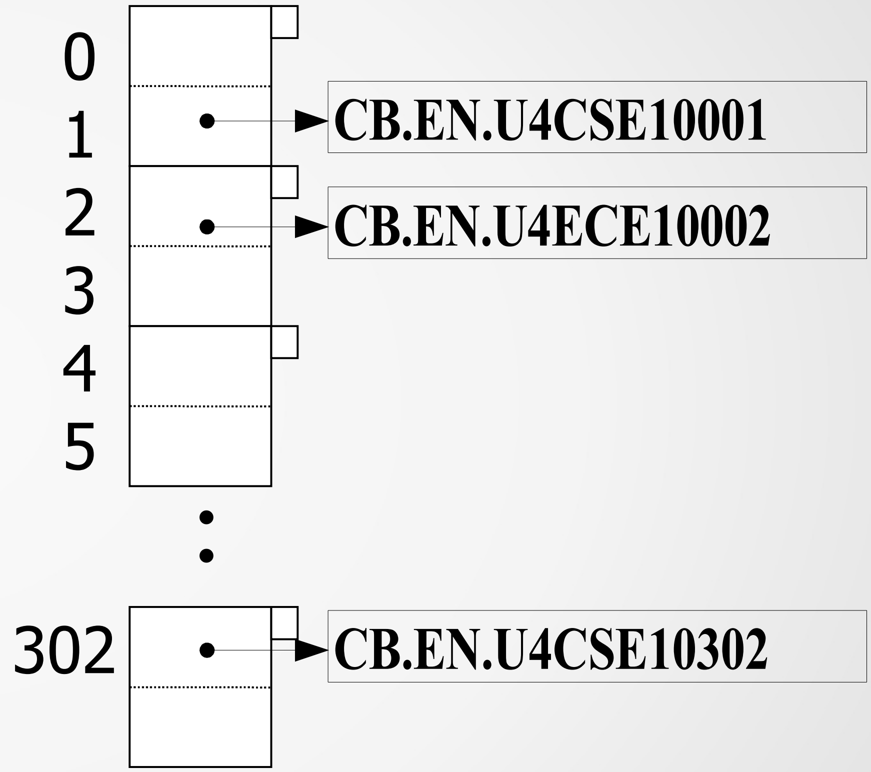
Index

Hash Function and Hash Tables

- Hash function *h*
  - Maps keys of a given type to integers in a fixed interval [0 to N-1]
  - e.g $h(x) = x \bmod N$
    - The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
  - goal is to store item (k, o) at index $i = h(k)$

Example

- Design a hash table for a dictionary storing items (Roll No, Name)
- Proposed hash table uses array of size N = 10,000 and the hash function h(x) = last three digits of x



Hash Functions

- A hash function specified as composition of two functions:
  - Hash code map
    - h1: keys → integers
  - Compression map:
    - h2: integers → [0, N − 1]
  - h(x) = h2(h1(x))
    - The values returned by a hash function are called hash codes or hash values

Hash Code Map

- Hash code map maps data to integers
  - This is important when using non integer data have to be hashed
  - They are converted to integers and then hashing is applied on these values
- Example
  - a string is converted to hash code by taking ASCII values of each letter and summing them
  - Memory addresses are reinterpreted as integers

Hash Code Maps

- Techniques to create hash codes
  - Integer casting
    - Suitable for keys whose length is lesser than that of integer
  - Component sum
    - Partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and sum the components
    - Suitable for keys whose length is greater than that of integer
  - Polynomial Accumulation
    - Create a polynomial using the partitions and evaluate it

Compression Maps

- A function to map the integer to some fixed range of values
- Division Method
  - $h2(y) = y \bmod N$
  - The size N of the hash table is usually chosen to be a prime
- Multiply, Add and Divide (MAD)
  - $h2(y) = (ay + b) \bmod N$
  - a and b are nonnegative integers such that $a \bmod N \neq 0$
  - Otherwise, every integer would map to the same value b

Hash Tables

- Insert(k,o): Insert object o with key k
  - Apply hash function h(k) e.g h(k) = k mod n
  - Insert 0 in bucket pointed by h(k)
- FindElement(k)
  - Let i be the bucket as a result of applying the hashing h(k)
  - Goto bucket i and search for k
- DeleteElement(k)
  - Find the bucket containing element using hash function and remove it

Collision Handling

- The goal of hashing is to map the keys randomly so that
  - Keys are distributed in the buckets evenly
  - Skews increase the worst case search complexity
- Collisions occur when different elements are mapped to the same cell
  - And if the bucket is full

Chaining

- Chaining:
  - Add a linked list to end of bucket and add additional elements to the linked list
  - Elements that hash to the same bucket are stored in a linked list
- Performance
  - Simple
  - Requires extra space, and search time increases

Chaining

INSERT:

$h(1) = 1$

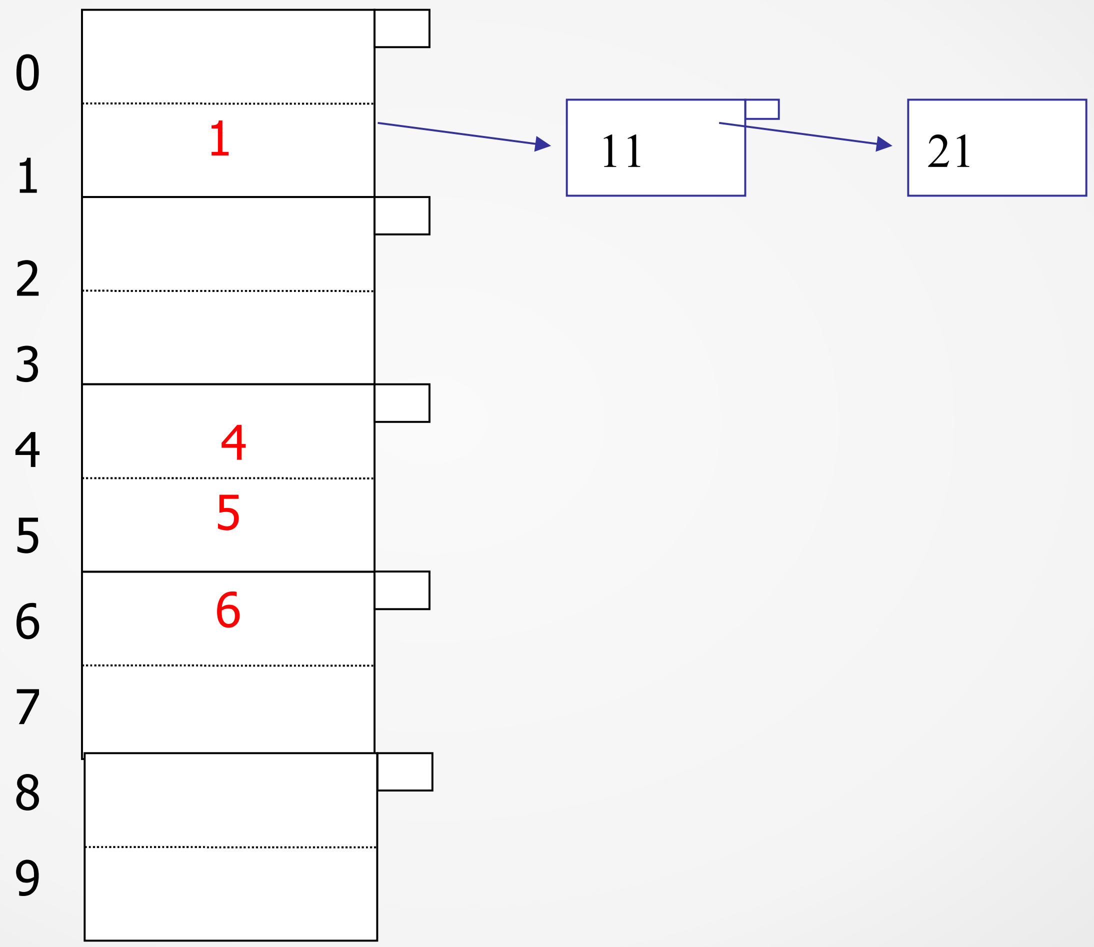
$h(6) = 6$

$h(5) = 5$

$h(4) = 4$

$h(11) = 1$

$h(21) = 1$



Open Addressing

- Chaining requires auxilliary data structures
- Open addressing
  - The colliding item is placed in a different cell of the table
- Linear probing
  - handles collisions by placing the colliding item in the next (circularly) available table cell
  - Each table cell inspected is referred to as a “probe”
  - Try to insert item into bucket A[i], where i = h(k).If collision occurs we try
    - A[(i+1) mod N], else A[(i+2) mod N] and so on until an empty bucket is found

Linear Probing

h=kmod 10

INSERT:

h(1) = 1

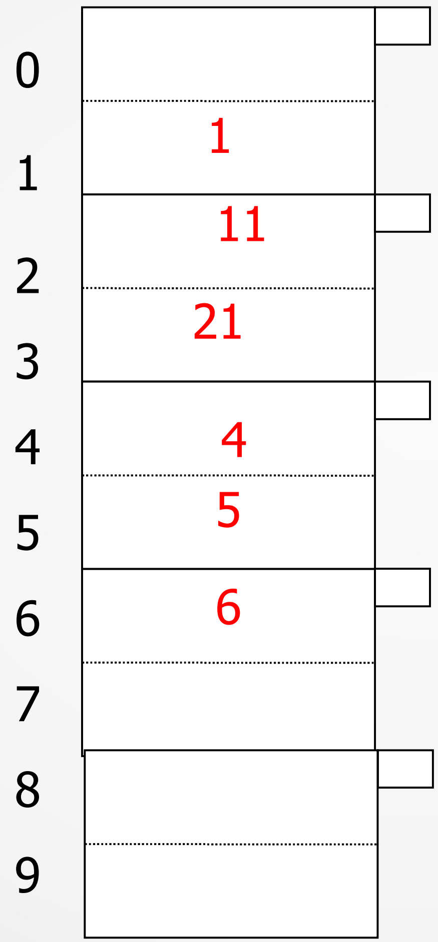
h(6) = 6

h(5) = 5

h(4) = 4

h(11) = 1

h(21) = 1



- **Search**
  - Start at cell h(k)
  - Probe consecutive locations until
    - An item with key k is found, or
    - An empty cell is found, or
    - N cells have been unsuccessfully probed

Update Operations

- When deleting an item, search becomes complex
  - Elements that have been placed using probing may have to be shifted after each delete so that search is not affected
- To handle insertions and deletions, we introduce a special object, called AVAILABLE, which replaces deleted elements removeElement(k)
- We search for an item with key k
  - If such an item (k, o) is found, we replace it with the special item AVAILABLE and we return the position of this item
  - Else, we return a null position

Quadratic Probing

- Probing occurs as follows
  - Buckets checked for space is A[(i+f(j)) mod N],for j = 0,1,2,.., where $f(j) = j^2$
- Avoids clustering patterns occuring in linear probing
- May not find a free space depending on the value of N

Quadratic Probing

INSERT:

$h(1) = 1$

$h(6) = 6$

$h(5) = 5$

$h(4) = 4$

$h(11) = 1$

$h(21) = 1$

0	21
1	1
2	11
3	
4	4
5	5
6	6
7	
8	
9	

- h(11)=1,hence collision
  - i+f(j) modN = (i+0)mod N =1
  - i+f(1) modN = (i+1)mod N = 2
- H(21)
  - i+f(j) modN = (i+0)mod N =1
  - i+f(1) modN = (i+1)mod N = 2
  - i+f(2) modN = (i+4)mod N = 5
  - i+f(3) modN = (i+9)mod N = 0

Double Hashing

- Uses a secondary hash function $h'(k)$ and handles collisions by placing an item in the first available cell of the series
  - $(i + f(j)) \bmod N$ where $f(j) = j.h'(k)$ for $j = 0, \ldots, N - 1$
  - $i=h(k)$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells

Double Hashing

- Common choice of compression map for the secondary hash function
  - $h'(k) = q - k \bmod q$ where
    - $q < N$
    - q is a prime
- The possible values for
  - $h'(k)$ are $1, 2, \ldots, q$

Double Hashing

INSERT:

h(1) = 1

h(6) = 6

h(5) = 5

h(4) = 4

h(11) = 1

h(21) = 1

0	
1	1
2	
3	
4	4
5	5
6	6
7	11
8	21
9	

- h'(k) =7 − k mod 7
- h(11)=1,hence collision
  - h'(11) = 7-11mod7=3
  - i+f(1) = (i+1.(3)) = 4
  - i+f(2) = (i+2.(3)) = 7
- h(21)
  - h'(21) = 7-21mod7=7
  - i+f(1) = (i+1.(7)) = 8

Hashing Performance

- Space utilization:

$$\text{Utilization} = \frac{\#\text{ keys used}}{\text{total }\#\text{ keys that fit}}$$

  - If too small: wasting space
  - If too big, overflows significant
- A good utilization
  - depends on how good the hash function is

Load Factor

- Load Factor: $\lambda = n/N$
  - n is number of stored elements
  - N is size of array of buckets
  - portion of the s buckets in the hash table that are filled with one of the n elements, and is preferable <1
    - Too low, space unused
    - Must be < 0.5 for open addressing schemes and <0.9 for separate chaining
    - If load factor increases beyond this, collision occurs frequently
- Impact on complexity
- Complexity proportional to $O(1+\lambda)$