Hashing - I

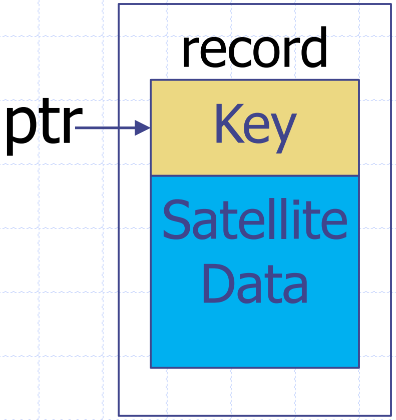
Instructor: Ashok Singh Sairam

Lecture Plan

- Dictionary – review
- Direct Hashing
- Hash tables
  - Hash function
  - Handling Collision
  - Average Search Time
- Hash Functions

Dictionary

- Abstract data type
- Maintain set of items each with a key (record)
- Supports the following operations
  - INSERT(x): Overwrites any existing key
  - DELETE(x): Delete record x
  - SEARCH(k): return record with key k or report that it doesn't exist



The Search Problem

- Find items with **keys** matching a given **search key**
  - *Given an array A, containing n keys, and a search key x, find the index i such as x=A[i]*
- Searching for an element in a linked list or array will take $\Theta(n)$
- Objective: Want to search in O(1) time in average case without the need of any <u>preprocessing</u>

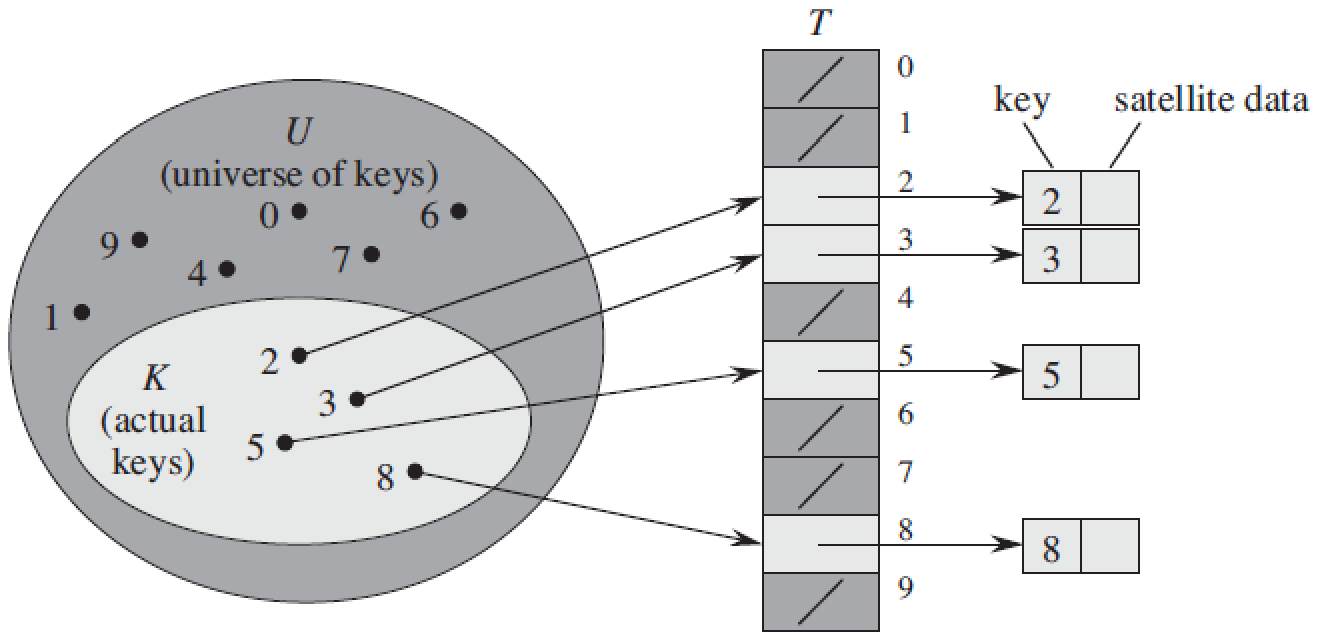
Applications

- Spellchecker
  - Search for word in a repository
- Keep track of reservations on flights
  - Search to find empty seats, cancel/modify reservations
- Search engine
  - Looks for all documents containing a given word

Direct Addressing

- Assumptions:
  - Key values are distinct
  - Each key is drawn from a universe $U = \{0, 1, \ldots, m - 1\}$
- Idea:
  - Store the items in an array, indexed by keys
- Direct-address table representation:
  - An array T[0 . . . m - 1]
  - Each **slot**, or position, in T corresponds to a key in U
  - For an element x with key k, a pointer to x (or x itself) will be placed in location T[k]
  - If there are no elements with key k in the set, T[k] is empty, represented by NIL

Direct Addressing: Implementation



Operations

DIRECT-ADDRESS-SEARCH(T, k)

1 **return** $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

1 $T[x.key] = x$

DIRECT-ADDRESS-DELETE(T, x)

1 $T[x.key] =$ NIL

- Running time for these operations: $O(1)$

Comparing Different Implementations

- Implementing dictionaries using:
  - Direct addressing
  - Ordered/unordered arrays
  - Ordered/unordered linked lists

	Insert	Search
direct addressing		
ordered array		
ordered list		
unordered array		
unordered list		

Comparing Different Implementations

- Implementing dictionaries using:
  - Direct addressing
  - Ordered/unordered arrays
  - Ordered/unordered linked lists

	Insert	Search
direct addressing	O(1)	O(1)
ordered array	O(N)	O(lgN)
ordered list	O(N)	O(N)
unordered array	O(1)	O(N)
unordered list	O(1)	O(N)

Examples Using Direct Addressing

- Example 1: Store student records of a class
  - #Students in a class at most ~400

- Example 2: Store records of citizens with key as Adhar number (12 digit key)
  - Size of Universe: |U|=Trillion, very large
  - Size of key: |K| = 12, relatively very small
  - Need a very large array, inefficient

Hash Tables

- When K is much smaller than U, a **hash table** requires much less space than a **direct-address table**
  - Can reduce storage requirements to $|K|$
  - Can still get $O(1)$ search time, but on the average case, not the worst case

Hash function

Idea:

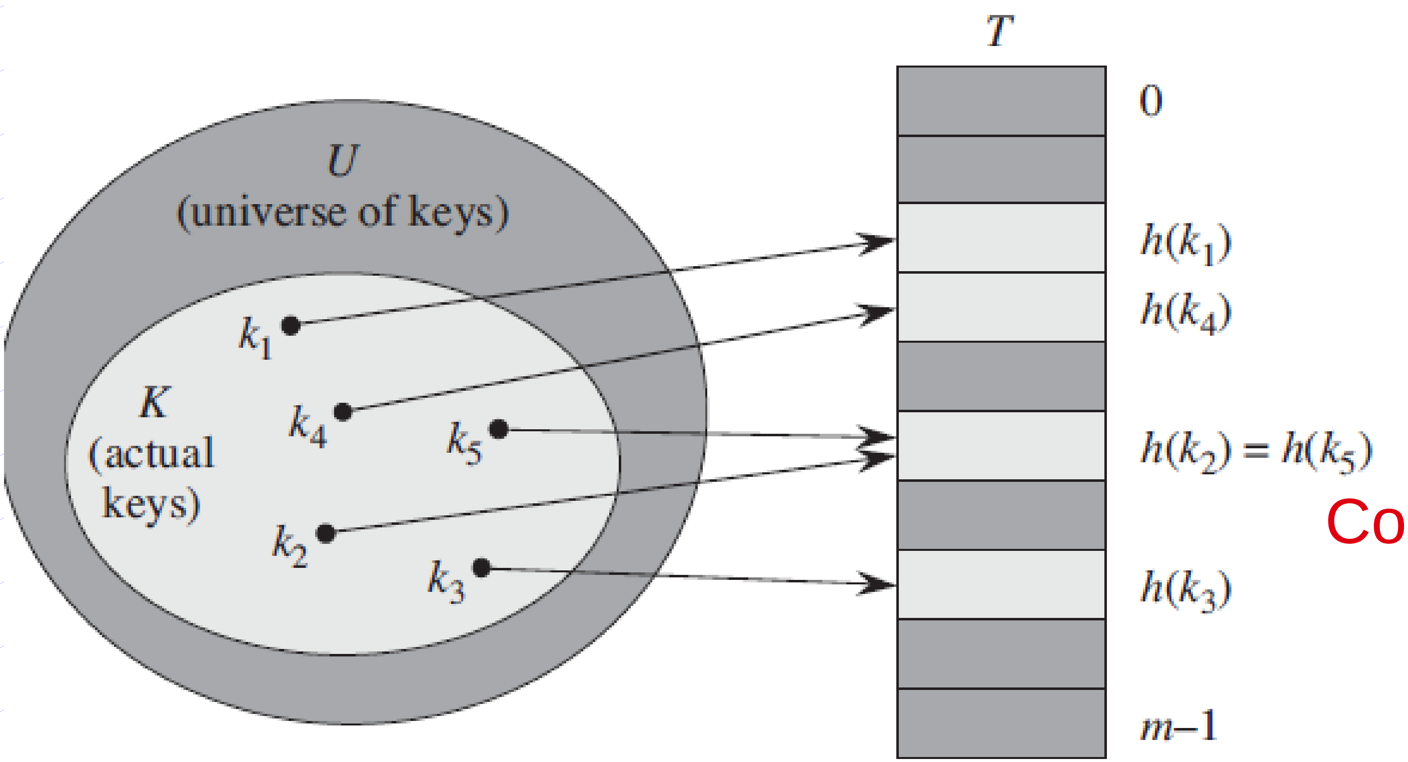
- Use a function h to compute the slot for each key
- Store the element in slot $h(k)$

- A **hash function** h transforms a key into an index in a hash table $T[0...m-1]$:

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

- We say that k **hashes** to slot $h(k)$
- Advantages:
  - Reduce the range of array indices handled: m instead of $|U|$
  - Storage is also reduced
- Disadvantage: Two keys may hash to the same slot (collision)

Hash function: Example



Collisions!

Collisions

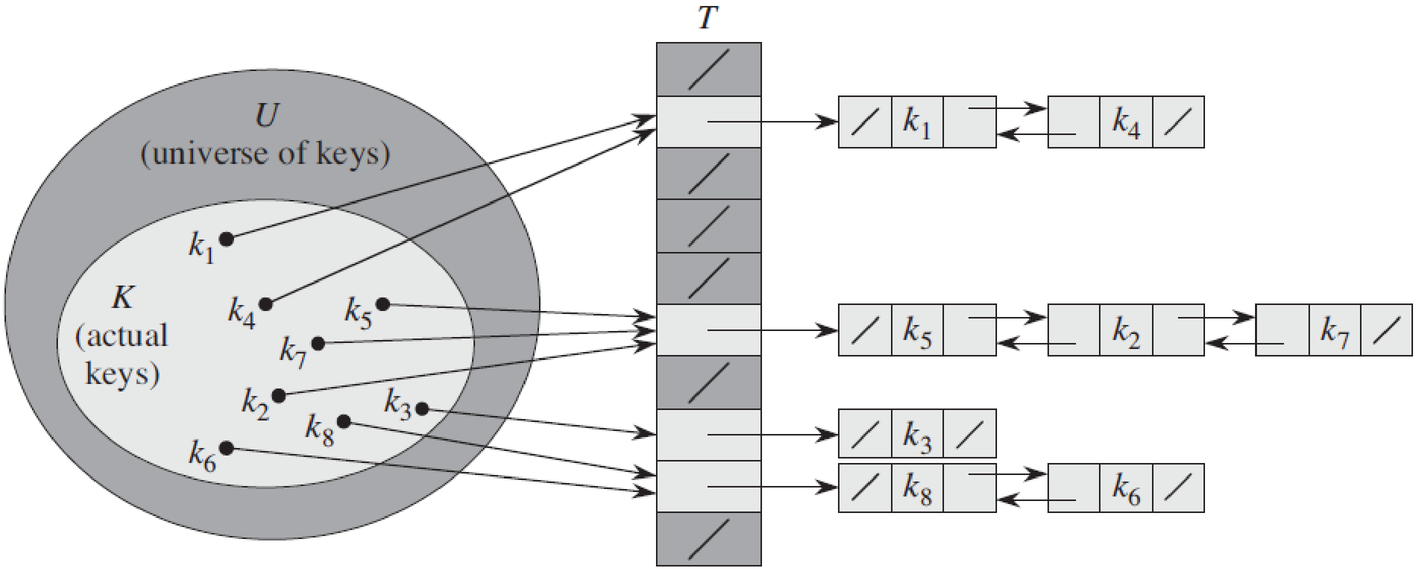
- Two or more keys hash to the same slot!!
- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value)
- Avoiding collisions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

Handling Collisions

- We will review the following methods:
  - Chaining
  - Open addressing
    - Linear probing
    - Quadratic probing
    - Double hashing
- We will discuss chaining first, and ways to build “good” functions.

Collision resolution by chaining

- Put all elements that hash to the same slot into a linked list



- Slot j contains a pointer to the head of the list of all elements that hash to j
  - Nil if there are no elements

Insertion in Hash Tables

CHAINED-HASH-INSERT(T, x)

insert x at the head of list $T[h(key[x])]$

- Worst-case running time is $O(1)$
- Assumes that the element being inserted isn't already in the list
- It would take an additional search to check if it was already inserted

Deletion in Hash Tables

CHAINED-HASH-DELETE(T, x)

delete x from the list $T[h(key[x])]$

- Need to find the element to be deleted.
- Worst-case running time:
  - Deletion depends on searching the corresponding list

Searching in Hash Tables

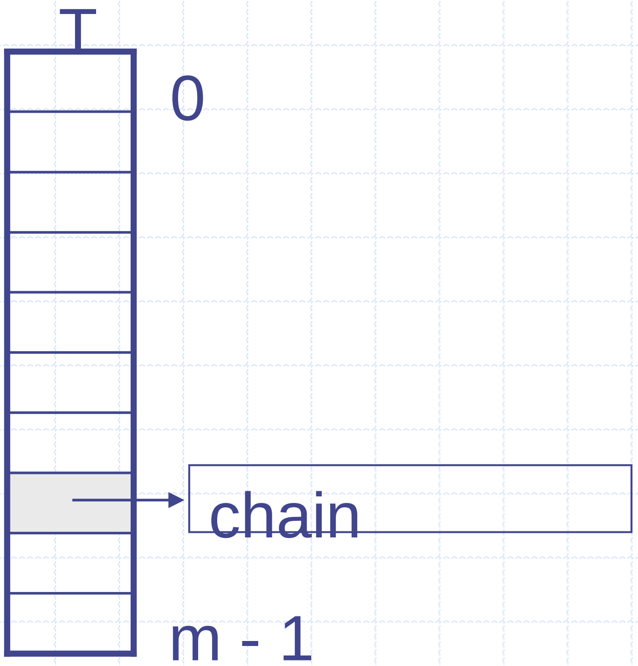
CHAINED-HASH-SEARCH(T, k)

search for an element with key k in list $T[h(k)]$

- Running time is proportional to the length of the list of elements in slot $h(k)$

Analysis of Hashing with Chaining: Worst Case

- How long does it take to search for an element with a given key?
- Worst case:
  - All n keys hash to the same slot
  - Worst-case time to search is $\Theta(n)$, plus time to compute the hash function

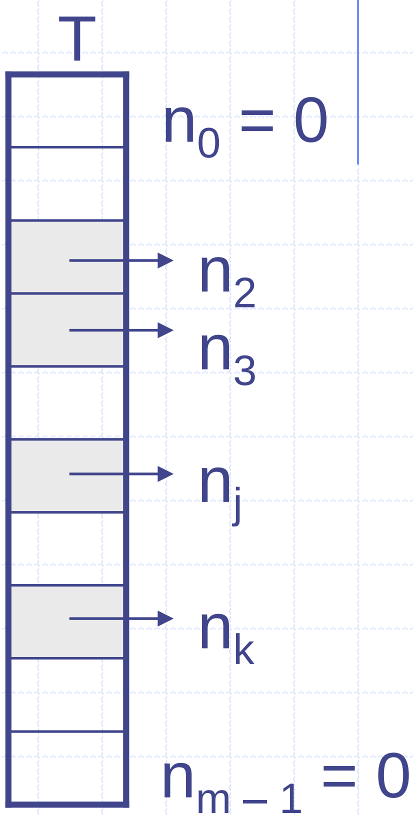


Analysis of Hashing with Chaining: Average Case

- **Assumption: Simple uniform hashing**
  - Any given element is equally likely to hash into any of the m slots
  - i.e., probability of collision Pr(h(x)=h(y)), is 1/m)
- Notation: n_j denote length of j^{th} entry,

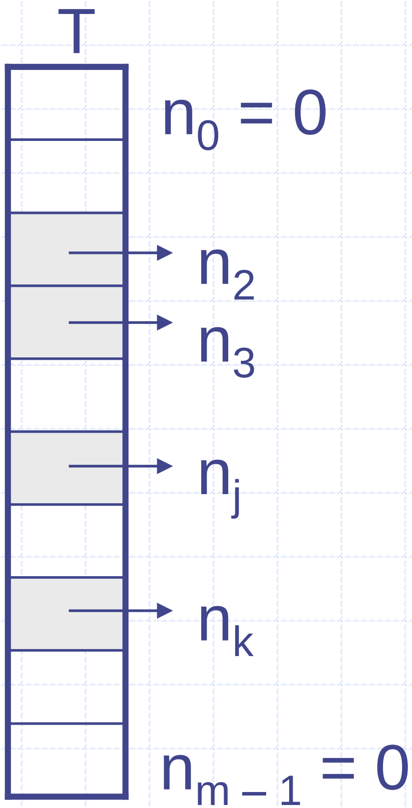
  [j= 0, 1, . . . , m – 1]

  : n denote total #keys
- Average value of n_j: (α)



Analysis of Hashing with Chaining: Average Case

- Average case
  - depends on how well the hash function distributes the n keys among the m slots
- #keys in the table: $n = n_0 + n_1 + \cdots + n_{m-1}$
- Average value of n_j: $E[n_j] = \alpha = n/m$



Load Factor of a Hash Table

- Load factor of a hash table T:

$$\alpha = n/m$$

  - n = # of elements stored in the table
  - m = # of slots in the table = # of linked lists
- α encodes the average number of elements stored in a chain
- α can be <, =, > 1

Unsuccessful search

Theorem: An unsuccessful search in a hash table takes expected time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing (i.e., probability of collision Pr(h(x)=h(y)), is 1/m)

Proof

Successful Search

- **Theorem: A** successful search in a hash table takes expected time $\Theta(1+\alpha)$ under the assumption of simple uniform hashing
- Proof: Search half of the list of length α plus O(1) time to compute h(k)

$$\Theta(1+\alpha)$$

Exercise

11.2-2

Demonstrate what happens when we insert the keys 5, 28, 19, 15, 20, 33, 12, 17, 10 into a hash table with collisions resolved by chaining. Let the table have 9 slots, and let the hash function be $h(k) = k \bmod 9$.

Acknowledgement

- Dr George Bebis, Foundation Professor, Dept of Computer Science and Engineering, University of Nevada Reno