Analysis of Algorithms

COMP.4040, Summer 2019

Chapter 11: Hash Table

By: Sirong Lin, PhD

University of Massachusetts Lowell

Learning with Purpose

Homework

HW 8: Due June 26 (W)

HW 10: Due June 24, 2019 (M)

Outline

Hash Tables Overview

Direct-address tables

Hash tables

- collision resolution by chaining
- collision resolution by open addressing

Hash functions

Hash Table Overview

Overview

Purpose:

maintain a (possible evolving) set of objects, perform super fast look-ups

Operations:

Insert, Delete, Look up (Search)

all through a key

Amazing guarantee: all operations expect O(1) time!

Overview (Cont'd)

universe of keys U, map these keys to a (look-up) table with many slots

Method 1: Direct Addressing

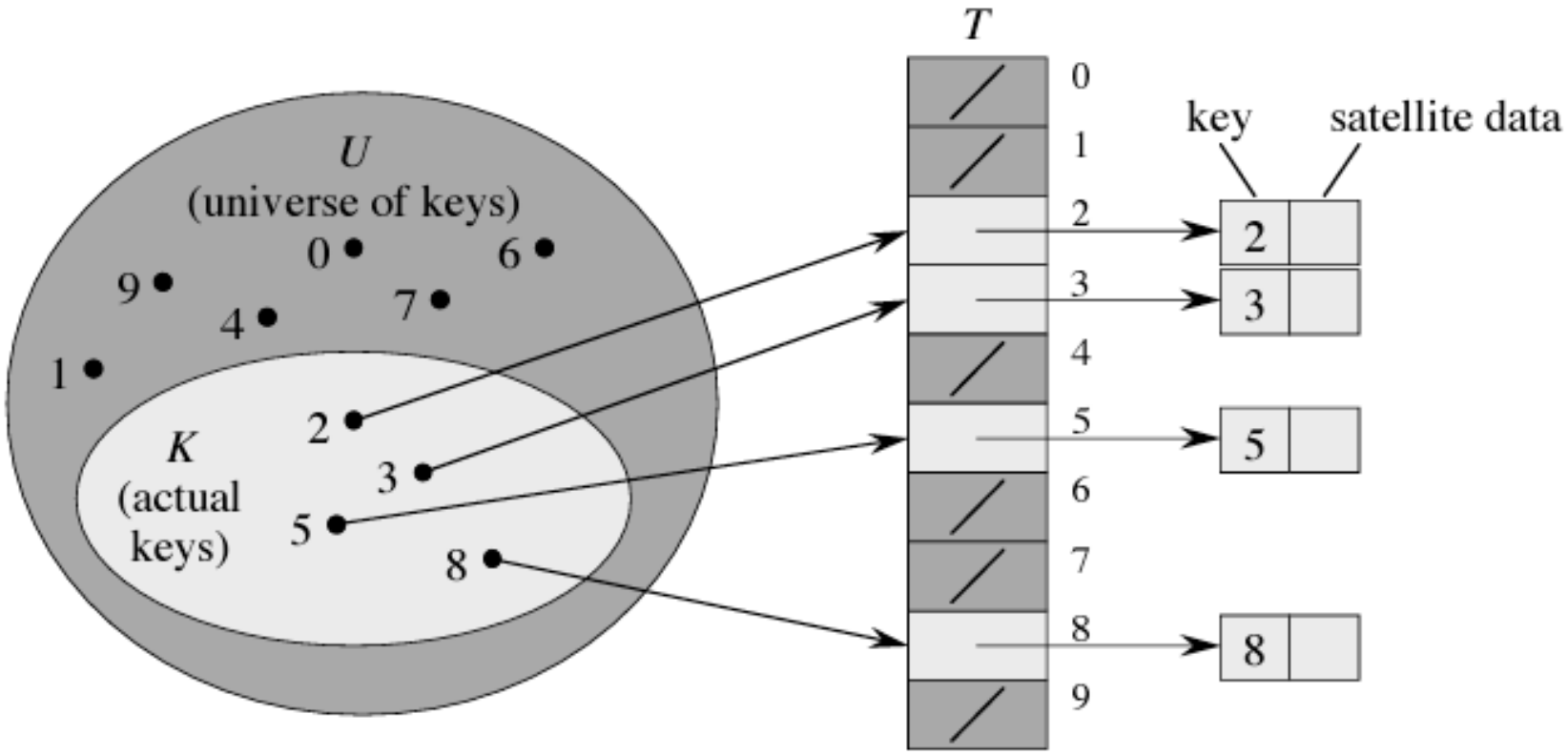
Method 2: Hash Table

Direct-address tables

Direct addressing: simple technique when the number of keys is reasonable small

- for a given key *k*, maps it into the k^{th} slot
- the size of the table (array) is the # of all possible keys

Direct-address tables (Cont'd)



Direct-address tables (Cont'd)

a **direct-address table** (array): T[0..m-1]

an element with key *k* is stored in slot *k*

m (not too large)

no two elements should have the same key

Direct-address tables (Cont'd)

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

Each operation takes O(1) time

Direct-address tables (Cont'd)

Downside:

- only works when m is small
- usually U is large though
- the set K of keys actually stored is small, most of the space allocated for the table T is wasted

Hash Table

often can't afford allocate one slot per possible key

- maintain a **hash** table, whose size much smaller than the universe U of all possible keys
  - typically proportional to the number of keys to be stored (rather than the actual # of possible keys)
- maps U of keys into the slots of a hash table T[0..m-1] (HOW?)

Hash Tables — Ideas

use a function to calculate slot number and store the element in slot h(k)

hash function *h*

h: U → {0, 1, …, m-1}, so that h(k) is a legal slot number in T

key *k* hashes to slot h(k)

h(k): hash value of key *k*

Hash Tables (Cont'd)

We have a large |U| (size of U) and a much smaller *m*, what will happen?

- pigeon hole principle — there are more than 1 key hashes to one slot
- when different keys map to the same slot, they collide

Hash Tables — Collision

Collision:

- when two or more keys hash to the same slot
- distinct keys x, $y \in U$, such that h(x) == h(y)
- can happen when there are more possible keys than slots (|U| > m)
  - we can design good hash function to minimize collisions, but it is hard to avoid

Hash Tables — Birthday Paradox

Consider *n* people with random birthdays (i.e., with each day of the year equally likely) how large does n need to be, before there is at least a 50% chance that two people have the same birthday?

A: 23 B: 57 C: 184 D: 367

the answer is A, this means that even with a small number like 23, it is very likely to get a collision

Hash Tables — Collision Solutions

Solution #1: Chaining

keep a linked list in each slot

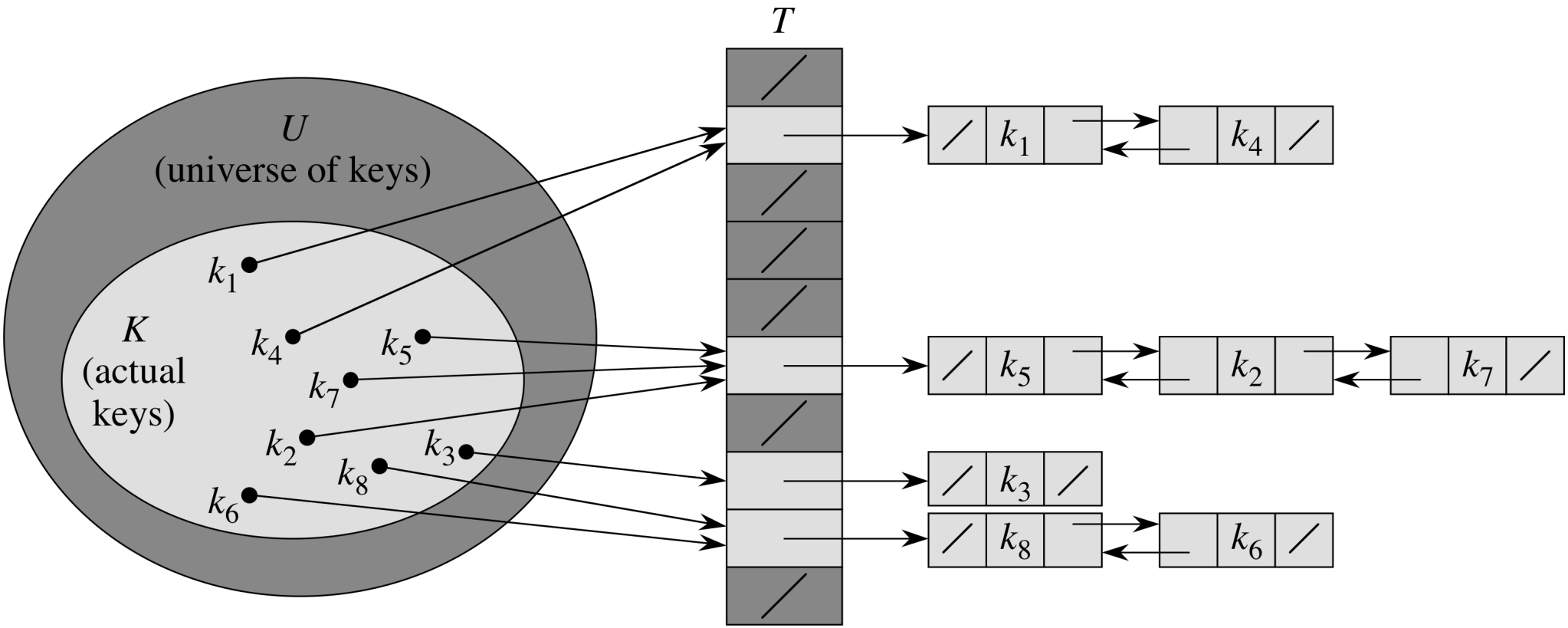
place all the elements that hash to the same slot into the same linked list

Solution #2: Open Addressing

only one object per slot

use probing to find an open slot

Hash table — Chaining



Collision resolution by chaining (doubly linked list)

Load factor

Load factor α = n/m

n: # of elements

m: # of slots in the table T (# of linked lists)

load factor is the average number of elements per linked list

$\alpha < 1$, $\alpha == 1$, $\alpha > 1$ (theoretically)

Hash table — Chaining (Cont'd)

Operations: Insertion, Search, Deletion

x is an element that we need to insert, search or delete from the hash table

Hash table — Chaining (Cont'd)

Insertion:

CHAINED-HASH-INSERT(T,x)

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

worst-case running time

O(1)

assumes that the element being inserted isn't already in the list

Hash table — Chaining (Cont'd)

Search:

CHAINED-HASH-SEARCH(T,k)

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

Hash table — Chaining (Cont'd)

If we have *n* elements, and a hash table *T* with *m* slots

how long does it take to find an element with that key?

or to determine that there is no element with that key?

running time

proportional to the length of the list of elements in slot h(k)

Hash table — Chaining (Cont'd)

Worst case

when all n keys hash to the same slot — forms a single list of length *n*

Average case

average length of the list: n/m (load factor)

depends on how well the hash function distributes the keys among the slots (on average)

Hash table —- Chaining (Cont'd)

Deletion:

CHAINED-HASH-DELETE(T,x)

delete an element *x* from the list T[h(key[x])]

x is the element, but not the key

assume that x is found already

if a doubly linked-list is used, worst-case running time is O(1)

Load factor (Cont'd)

What is the possible value of load factor α when we use a Chaining hash table implementation strategy?

>1

e.g., $\alpha = 2$, put 2000 objects into 1000 slots, each slot chains 2 objects on average

Analysis of hashing with Chaining (Cont'd)

Expected # of keys in each slot

$\alpha = n/m$ (the load factor)

Running time for chaining (search)

the time to compute the hash function and the time to search the linked list

$\Theta(1+\alpha)$

Analysis of hashing with Chaining — Control Load

Upshot I: For good hash table performance, need to control the load factor

$\alpha = O(1)$ is a necessary condition for operations to run in constant time

Analysis of hashing with Chaining — Control Load (Cont'd)

so should monitor the changes of α

- if #objects increase, increase # of slots (so we don't have too many collisions)
- if spaces is a concern, when there are lots of deletion, shrink the hash table size

Analysis of hashing with Chaining — Control Load (Cont'd)

However, α , the chaining list length, could be any where from n/m ~ n

n/m: equal-length lists

n: all keys in the same slot

Analysis of hashing with Chaining — good hash function

Upshot II: For good hash table performance, need a good hash function, so that objects spread evenly across slots

Analysis of hashing with Chaining — good hash function (Cont'd)

Ideally, the hash function satisfies the assumption of **simple uniform hashing**

- each key is equally likely to be hashed to any slot of the table, independent of where other keys hashing
- in practice, not possible to satisfy this assumption

no perfect hash functions either

Analysis of hashing with Chaining — good hash function (Cont'd)

Exercise: if we use simple uniform hashing (each key is equally likely to be hashed to any slot of the table, independent of where other keys hashing)

What is the probability of two keys k_i and k_j to have the same slot number, i.e., Pr{$h(k_i) == h(k_j)$}, $1 \leq i, j \leq n, (i \neq j)$? (assuming we have n keys, m slots)

Hash functions

Hash functions

General rule of good hash functions:

- easy to evaluate/store, e.g., mod
- try to satisfy the simple uniform hashing
  - derive the hash value in a way that independent of any patterns that might exist in the data

Hash functions — Division Method

e.g., h(k) = k mod m

m = 20 and k = 91, h(k) = 91 mod 20 = 11

advantage: fast and simple, one division operation

disadvantage: may easily produce collision if m is not good

Hash functions — Keys

Keys are natural numbers

- hash functions assume that keys are natural numbers
- convert (non-number) keys to numbers, e.g., CLRS
  - use their ASCII values: C = 67, L = 76, R = 82, S = 83
  - CLRS: $(67 \times 128^3) + (76 \times 128^2) + (82 \times 128^1) + (83 \times 128^0) = 141,764,947$

Hash functions — bad functions, Example 1

keys are SSN (9-digit numbers in three fields)

- first field identifies the geographical area where the number was issued and the other two fields identify the individuals
- suppose our application needs to process a few hundreds of keys, so the hash table m = 1000

Hash functions — bad functions, Example 1 (Cont'd)

keys are SSN (9-digit numbers in three fields)

- bad idea: use the first 3 digits
- better idea: use 9 digits as the key, and then use modular hashing (division method)

Hash functions — bad functions, Example 2

Keys are memory addresses (multiple of 2)

hash function: h(k) = k mod 2 (k is an address)

What will happen?

all odd buckets guaranteed to be empty

Hash functions — choose m

How to choose *m* (m denotes the # of slots)

- choose m to be a prime number (no common factor with the integers need to hash)
- not too close to a power of 2
- not too close to a power of 10

Comparison of using different m

keys are memory locations, addresses divisible by 4

key	mod 8	mod 7
0	0	0
4	4	4
8	0	1
12	4	5
16	0	2
20	4	6
24	0	3
28	4	0

Hash functions — Example for Chaining

Example: insert keys 7, 18, 11, 5, 26 in the order given into a hash table of length 5 using hash function $h(k) = k^3 \bmod m$, with chaining solution

see notes

Open Addressing

Open Addressing

An alternative for handling collisions

Idea:

each slot contains either a key or NIL

when a new key collides, find an empty slot through **probing**

Open Addressing — Probing

Probing Strategy:

- Linear probing
- Quadratic probing
- Double hashing

Open Addressing — Linear Probing

Linear Function: h(k, i) = (h'(k) + i) mod m

h'(k) is an auxiliary hash function

starts at slot h'(k), and continue to probe though the table

wrapping after slot *m-1* to slot 0

Open Addressing — Linear Probing (Cont'd)

Example:

insert keys 7, 18, 11, 5, 26 in the order given into a hash table of length 5 using hash function $h'(k) = k^3$, with Open Address — Linear Probing solution

Open Addressing — Quadratic Probing

Quadratic Function:

$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, where c_1 and c2 are non-zero constants. (i = 0, 1, …, m-1)

h'(k) is an auxiliary hash function

Open Addressing — Quadratic Probing (Cont'd)

Example

insert keys 7, 18, 11, 5, 26 in the order given into a hash table of length 5 using hash function $h'(k) = k^3$, $c_1 = 1$, $c_2 = 2$, with Open Address — Quadratic Probing solution

Open Addressing — Quadratic Probing (Cont'd)

may get secondary clustering

- if two distinct keys have the same h' value, then they have the same probe sequence
- i.e., $c_1 i + c_2 i^2$ will be the same for these keys

Open Addressing — Double Hasing

Double hashing:

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

$h_2(k)$ relatively prime to m (no factors in common other than 1) to ensure the probe sequence is a full permutations of <0, 1, …, m-1>

Open Addressing — Operations

Insert a key k

Search a key k

Delete a key k

Open Addressing — Search Operation

Search for key k

compute h(k) and examine slot h(k) — ***probe***

successful search: if slot h(k) contains key k

unsuccessful search: if the slot contains NIL

are we done? other cases?

Open Addressing — Search Operation (Cont'd)

if slot h(k) contains a key that is not k:

compute a new index based on different probing strategies;

keep probing until we find the key (successful search) or NIL (unsuccessful search)

Open Addressing — Delete Operation

- Delete a key
  - Can we just delete a key directly and replace it with a NIL?
    - NO
    - may generate wrong search result
  - Chaining is used more often because of this

Chapter Summary

What's the purpose of using Hash Tables?

Generate hash tables (space-time balance)

- map keys to hash table directly when space is not a concern, no collision, direct-address tables
- map keys using hash functions when space is limited — collision
  - chaining
  - open addressing

Chapter Summary (Cont'd)

Load factor

Hash functions

- bad and good
- calculate slot # using hash functions