Analysis of Algorithms 1 (Fall 2013) Istanbul Technical University Computer Eng. Dept.

Chapter 11

Hash Tables

Last updated: November 11, 2009

Purpose

Introduce dictionary

Introduce hash table as a dictionary implementation

Introduce methods of collision resolution in hash tables

Outline

- Dictionary
- Direct Access Table
- Hash Table
  - concept
  - collision resolution
  - choosing a hash function
  - advanced collision resolution

Dictionary

- Dictionary: a dynamic set that supports only INSERT, SEARCH, and DELETE
- Hash table: an effective data structure for dictionaries
  - O(n) time for search in worst case
  - O(1) expected time for search

Applications: Any application that requires fast search and/or insert, e.g. actual dictionary, employee database, image/song/document database... etc.

Dictionary

A dictionary includes

- Unique identification code (key)
- Additional data (satellite data)
- Operations to work on keys
- Operations
  - based on *equality*
  - *min, max, successor, predecessor* are not musts

Dictionary

A dictionary supports three functions (methods):

Search(T, k)

returns a pointer x to an element where k = x.key

Insert(T, x)

adds the element pointed to by x to T

Delete(T, x)

removes the element pointed to by x from T

Dictionary

Could be implemented using different data structure

- Arrays
- Linked lists
- Hash tables (This class)
- Binary trees
- Red/Black trees
- AVL trees
- B-trees

Direct Address Tables

Assume that the set of keys are drawn from the set
$U = \{0, 1, \ldots, m-1\}$
Set up an an array (table) $T\,[0\ldots m\text{-}1]$ of m elements

$$T[k] = \begin{cases} x & if \quad key[x] = k \\ nil & otherwise \end{cases}$$

If no two elements have the same key each search operation takes constant time

Problem:

The range of keys, i.e./ the size of the table can be huge:

- e.g.
  - 64-bit numbers (which represent 18,446,744,073,709,551,616different keys),
  - character strings (even larger!).

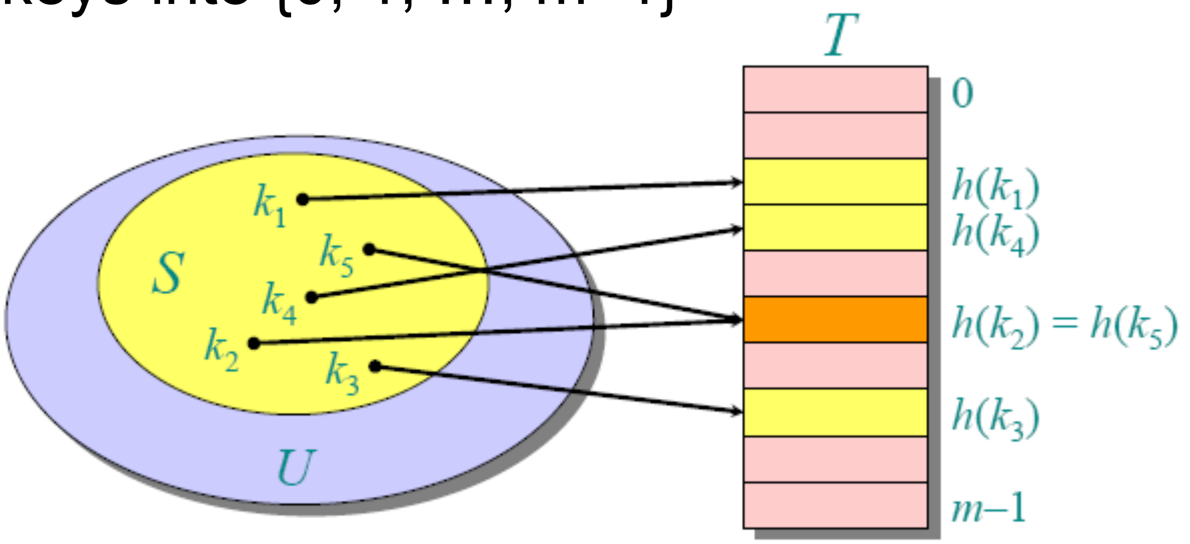
Hash Function

Problem:

The range of keys is huge

Solution:

Use a hash function h to map the universe U of all keys into {0, 1, …, m–1}



Hash Function

- **direct addressing:** an element with key k is stored in slot k.
- **hashing:** this element is stored in slot h(k);
- we use a **hash function h** to compute the slot from the key k.
- h maps the universe U of keys into the slots of a hash table T[0…m - 1]:

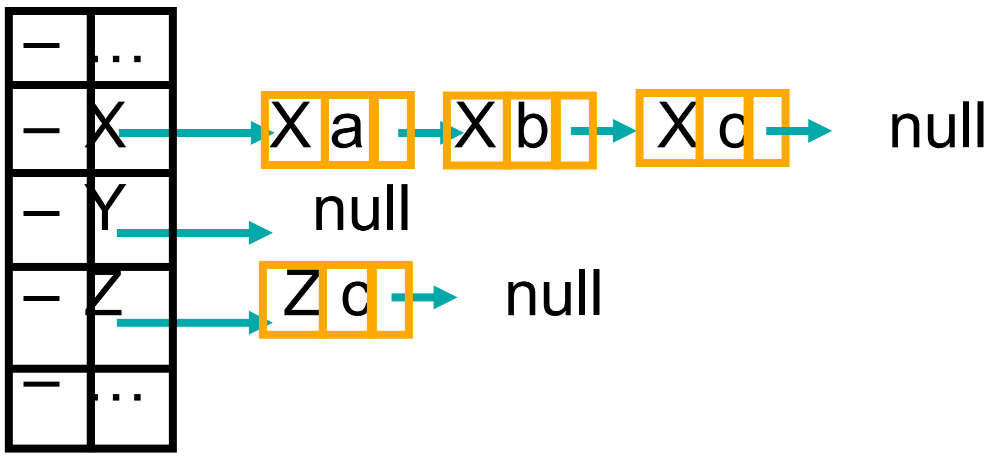
Collision Resolution

Collision: Two keys which have the same hash value

Collision is a problem.

A solution: Chaining

an array of links or a linked list to keep the elements having the same key



Analysis of Hashing (with Chaining)

- An element with key k is stored in slot $h(k)$
- $h(k)$ maps the universe of keys into the slots of hash table of length m
- m is reasonably small
- computation time of $h(k)$ is $\Theta(1)$

- Uniform hashing: hashed keys are expected to be distributed into slots equally

- Load factor (of a table): $\alpha = n/m$
- where
- m is # of slots
- n is # of elements being hold

Analysis of Hashing (with Chaining)

- Searching for an element with a key *k*
  - compute *h(k)*
  - locate its slot in the hash table
  - search for the element in the linked list
  - if the given element is in the linked list
    - return the element (with satellite data)
    - Otherwise unsuccessful search
- Average list length for uniform hashing: $\alpha = n/m$
  - i.e., expected number of elements to be visited

Analysis of Hashing (with Chaining)

- Search time for an unsuccessful search:
- O(1+ α)
  - Why: 1:compute the h(k),
    $\alpha = E[n_{h(k)}] = E$ length of list where h(k) is mapped

- Brief analysis of a successful search
  - a new element is inserted at the end of the linked list following a successful search
    - (think of searching algorithm and discuss the use of a tail pointer)
  - the expected length of the list before the insertion of the i^{th} element is (i-1)/m

Analysis of Hashing (with Chaining)

- The expected number
- of elements examined is

$$E\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}Xij\right) = \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[Xij]\right)$$

Xij=I[h(ki)=h(kj)]
Indicator random variable
showing if hash function outptus
The same value for ki and kj
If simple uniform hashing *Pr[h(ki)=h(kj)]=1/m*
E[Xij]=1/m (by Lemma 5.1.)

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) = 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$= 1+\frac{1}{nm}\cdot\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)$$

$$= 1+\frac{1}{nm}\cdot\left(n*n-\frac{n(n+1)}{2}\right)$$

$$= 1+\frac{n-1}{2m}$$

$$1+\frac{\alpha}{2}-\frac{1}{2m}$$

- Considering the time
- for computing the hash function

$$\Theta(2+\alpha/2-1/2m)=\Theta(1+\alpha)$$

Analysis of Hashing (with Chaining)

If the number of hash table slots is at least proportional to the number of elements in the table

$n=O(m)$

and

$\alpha = n/m$

$\alpha = O(m)/m=O(1)$

- searching takes constant time on average
- insertion takes O(1) worst-case time
- deletion takes O(1) worst-case time when the lists are doubly-linked
- ALL DICTIONARY OPERATIONS CAN BE SUPPORTED IN O(1) TIME!

Hash Functions

- A good hash function
  - quick to compute
  - distributes keys uniformly: i.e. Each key is equally likely to hash any of the m slots.

- Hashing non-integer keys
  - turning the keys into integers
    - remove hyphen or stroke
    - add up the ASCII values of the characters
  - then use a standard hash function on the integers

Hash Functions

- Division Method:
  - h(k) = k mod m
  - k: key, m: hash table size

Do not choose $m = 2^p$ (you will take lower order bits
a prime far from a power of 2 is good

- Multiplication Method:
  - h(k) = floor(m(k A mod 1))
  - A a constant, 0 <A < 1
  - k A mod 1 == fractional part of kA = kA – floor(kA)

Knuth suggests A=(sqrt(5)-1)/2=0.618033…

- Universal Hashing:
  - select hash functions at random (independent of the keys) from a carefully designed set of functions at the beginning of execution.
  - Can yield provably good performance on average (see pp 233)

Resolving collisions by open addressing

Open addressing: No storage is used outside of the hash table itself.

(compare to chaining where linked lists are stored)

Insertion systematically probes the table until an empty slot is found.

The hash function depends on both the key and probe number:

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

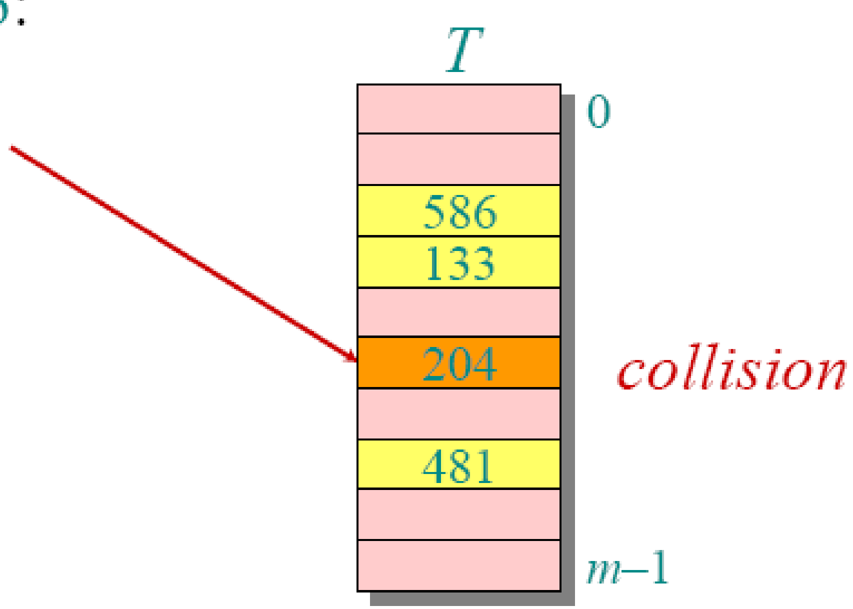
The probe sequence $\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$ should be a permutation of $\{0, 1, \ldots, m-1\}$.

The table may fill up, and deletion is difficult (but not impossible).

Open addressing example

Insert key $k = 496$:

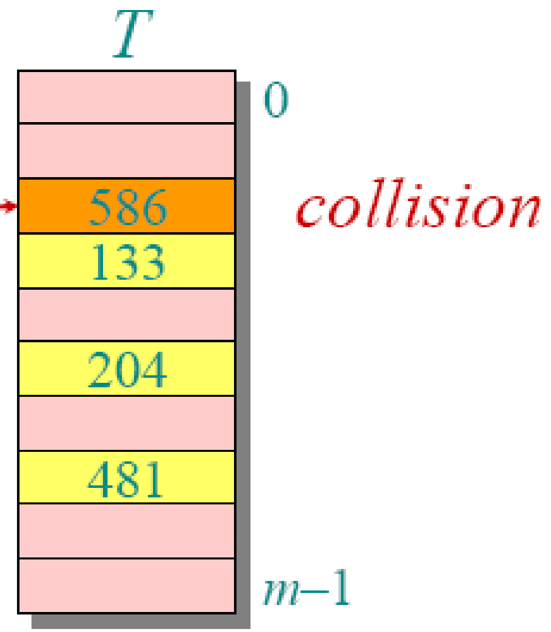
0. Probe $h(496,0)$



Open addressing example

Insert key $k = 496$:

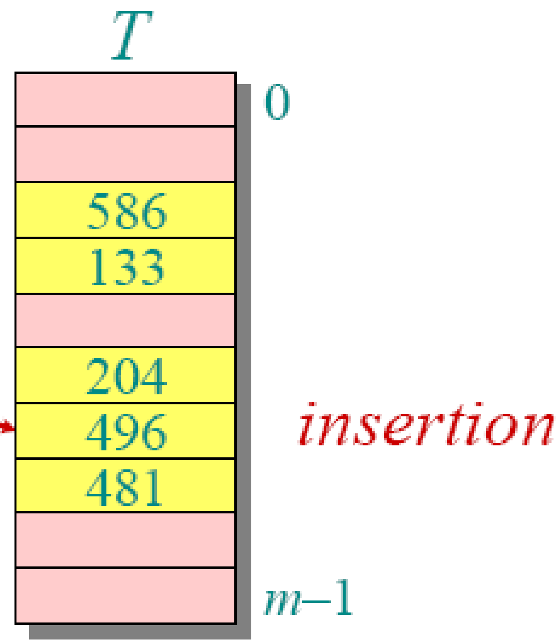
0. Probe $h(496,0)$
1. Probe $h(496,1)$



Open addressing example

Insert key $k = 496$:

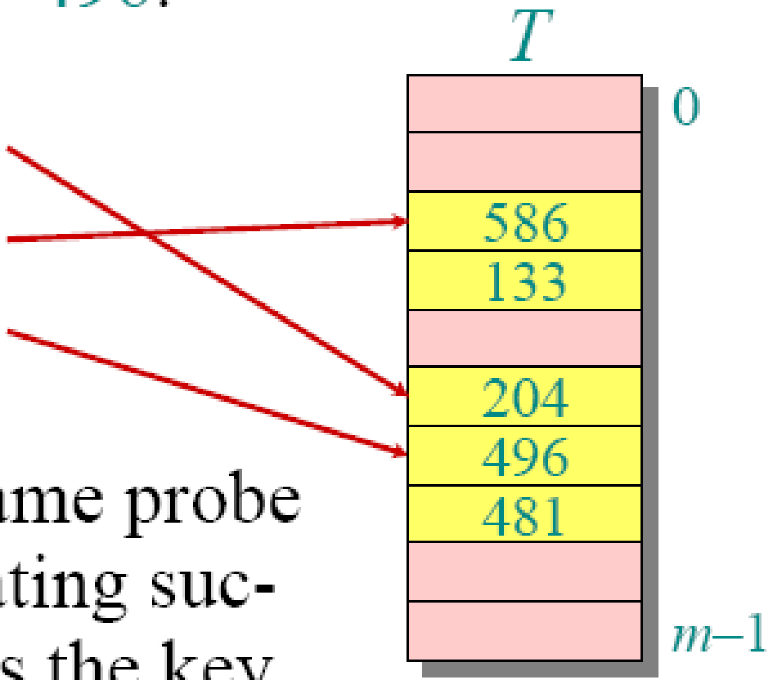
0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$



Open addressing example

Search for key $k = 496$:

0. Probe $h(496,0)$
1. Probe $h(496,1)$
2. Probe $h(496,2)$



Search uses the same probe sequence, terminating successfully if it finds the key and unsuccessfully if it encounters an empty slot.

Probing Strategies

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

- Quadratic Probing
- $h(k,i) = (h'(k)+c_1 i + c_2 i^2) \bmod m$

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

Linear Probing

- Given an ordinary hash function $h'(k)$, linear probing uses the hash function
  - $h(k,i) = (h'(k) + i) \bmod m$.
- This method, suffers from **primary clustering**, where long runs of occupied slots build up, increasing the average search time. Moreover, the long runs of occupied slots tend to get longer.

Double hashing

Given two ordinary hash functions $h_1(k)$and $h_2(k)$, double hashing uses the hash function

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

This method generally produces excellent results, but $h_2(k)$ must be relatively prime to m. One way is to make ma power of 2 and design h (k) to produce only odd numbers.

Example

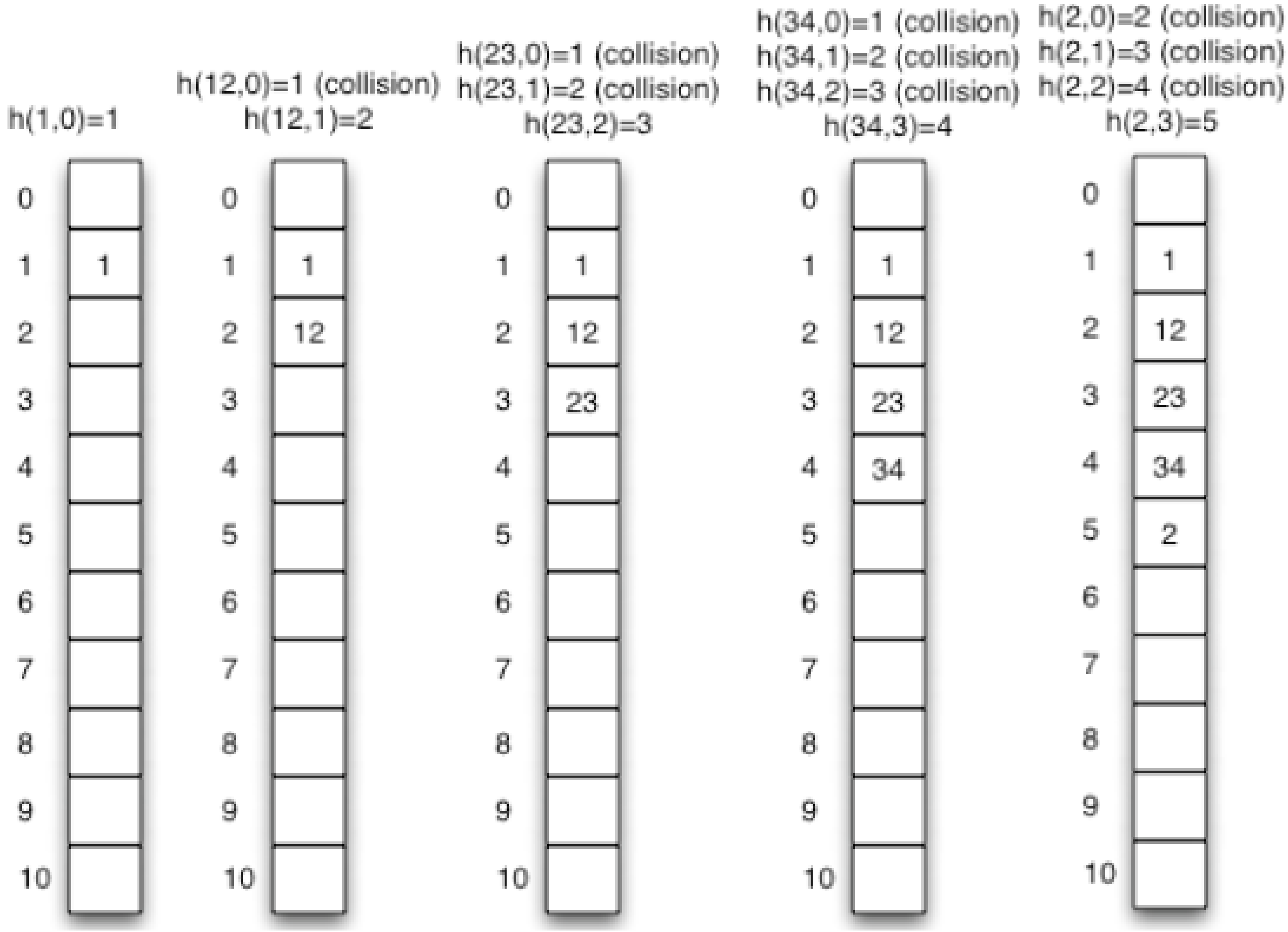
Insert B={1,12,23,34,2}

into a hash table of size m=11,

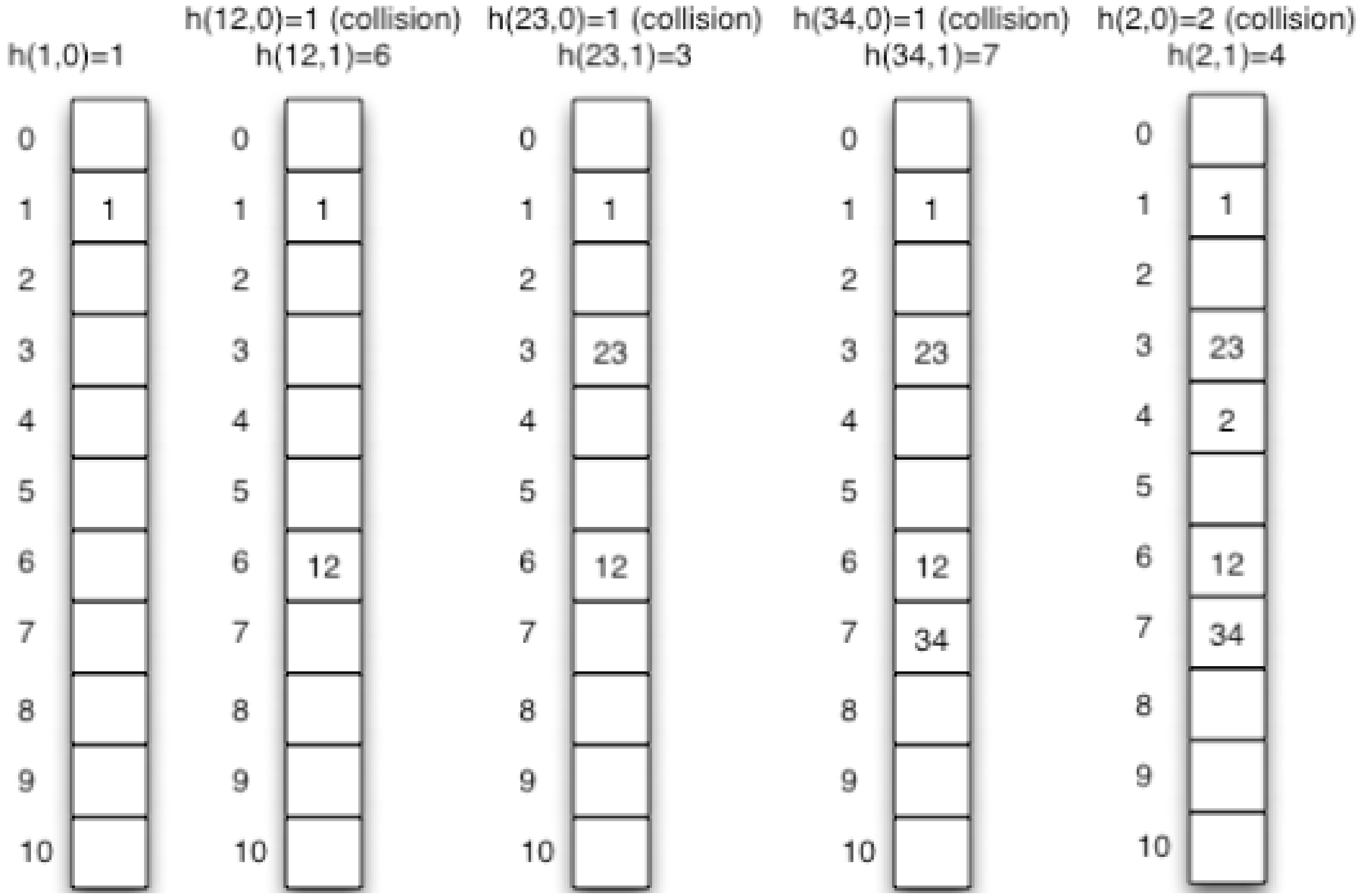
a) using linear probing

b) Using double hashing with h2(k)=(k mod 7)

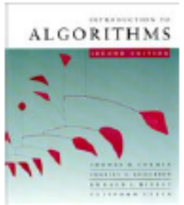
Example: linear probing



Example: double hashing



Universal Hashing

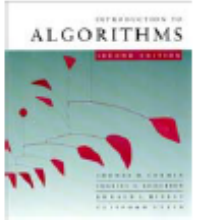


A weakness of hashing “as we saw it”

Problem: For any hash function h, a set of keys exists that can cause the average access time of a hash table to skyrocket.

- An adversary can pick all keys from $h^{-1}(i) = \{k \in U : h(k) = i\}$ for a slot i.
- There is a slot i for which $|h^{-1}(i)| \geq u/m$

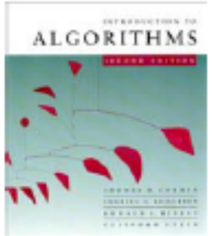
Universal Hashing



Solution

- Randomize!
- Choose the hash function at random from some family of function, and independently of the keys.
- Even if an adversary can see your code, he or she cannot find a bad set of keys, since he or she doesn't know exactly which hash function will be chosen.
- What family of functions should we select ?

Universal Hashing



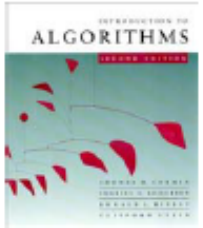
Family of hash functions

- Idea #1: Take the family of ***all*** functions

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

  That is, choose each of $h(0)$, $h(1)$, … , $h(u\text{-}1)$ independently at random from $\{0 \ldots m\text{-}1\}$
- Benefit:
  - The uniform hashing assumption is true!
- Drawback:
  - We need u random numbers to specify h. Where to store them ?

Universal Hashing



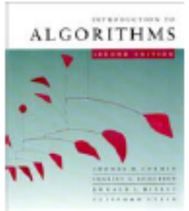
Universal hashing

Idea #2: Universal Hashing

- Let $\mathcal{H}$ be a finite collection of hash functions, each mapping U to $\{0, 1, \ldots, m-1\}$
- We say $\mathcal{H}$ is ***universal*** if for all $x, y \in U$, where $x \neq y$, we have

$$\Pr_{h \in \mathrm{H}}\{h(x) = h(y)\}| = 1/m.$$

Universal Hashing



Constructing a set of universal hash functions

- Let m be prime.
- Decompose key k into $r + 1$ digits, each with value in the set $\{0, 1, \ldots, m-1\}$.
- That is, let $k = \langle k_0, k_1, \ldots, k_r \rangle$, where $0 \le k_i < m$.

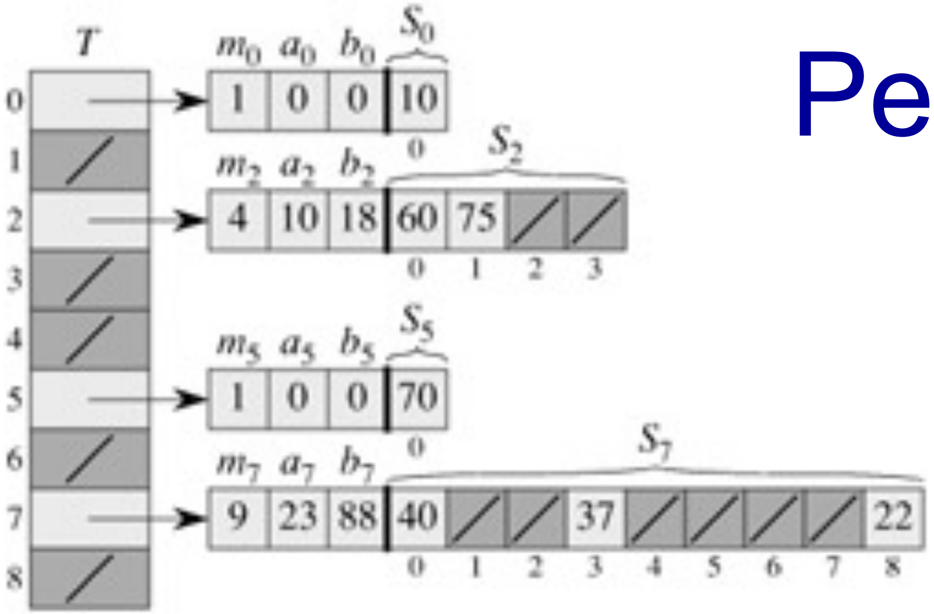
Randomized strategy:

- Pick $a = \langle a_0, a_1, \ldots, a_r \rangle$ where each a_i is chosen randomly from $\{0, 1, \ldots, m-1\}$.
- Define $h_a(k) = \sum_{i=0}^{r} a_i k_i \bmod m$
- Denote $H = \{h_a \colon a \text{ as above}\}$

Perfect Hashing

- perfect hashing if the worst-case number of memory accesses required to perform a search is O(1).
- Use a two level hashing scheme with universal hashing at each level.

Perfect Hashing



Using perfect hashing to store the set *K = {10, 22, 37, 40, 60, 70, 75}. The outer* hash function is *h(k) = ((ak + b) mod p) mod m, where a = 3, b = 42, p = 101, and m = 9. For* example, *h(75) = 2, so key 75 hashes to slot 2 of table T . A secondary hash table Sj stores all* keys hashing to slot *j . The size of hash table Sj is mj , and the associated hash function is hj* (*k*) *= ((aj k + bj) mod p) mod mj. Since h2(75) = 1, key 75 is stored in slot 1 of secondary hash* table *S2. There are no collisions in any of the secondary hash tables, and so searching takes* constant time in the worst case.

Summary

Dictionary

Direct Access Table

Hash Table

- concept
- collision resolution
- choosing a hash function
- advanced collision resolution