HashTable
CISC4080, Computer Algorithms
CIS, Fordham Univ.

Instructor: X. Zhang

Spring 2018

Acknowledgement

- The set of slides have used materials from the following resources
  - Slides for textbook by Dr. Y. Chen from Shanghai Jiaotong Univ.
  - Slides from Dr. M. Nicolescu from UNR
  - Slides sets by Dr. K. Wayne from Princeton
    - which in turn have borrowed materials from other resources
  - Other online resources

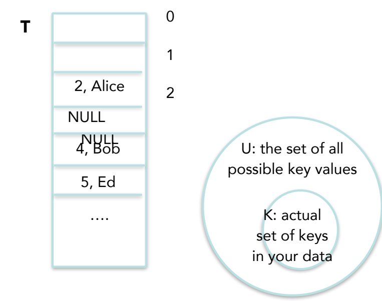
Support for Dictionary

- **Dictionary ADT**: a dynamic set of elements supporting INSERT, DELETE, SEARCH operations
  - elements have distinct key fields
  - DELETE, SEARCH by key
- Different ways to implement Dictionary
  - unsorted array
    - insert O(1), delete O(n), search O(n)
  - sorted array
    - insert O(n), delete O(n), search O(log n)
  - binary search tree
    - insert O(log n), delete O(log n), search O(log n)
  - linked list …
- Can we have "almost" constant time insert/delete/search?

Towards constant time

- Direct address table: use key as index into the array
  - T[i] stores the element whose key is i

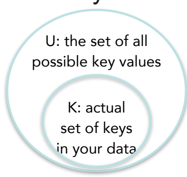
Insert (element(2,Alice))
  T[2]=element(2, Alice);
Delete (element(4))
  T[4]=NULL;
Search (element(5))
  return T[5];

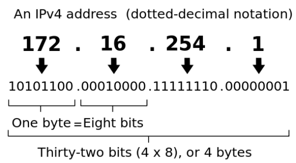


- How big is the table?
  - big enough to have one slot for every possible key

Case studies

- A web server: maintains all active clients' info, using IP addr. as key

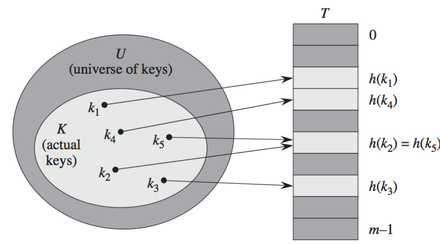




- Universe of keys: the set of all possible IPv4 addr., $|U|=2^{32}$
  - much bigger than total # of active clients
  - Too big to use direct access table:
    - a table with 2^{32} entries, if each entry is 32bytes, then 128GB is needed!
- How to have constant accessing time, while not requiring huge memory usage?

Hash Table

- Hash Table: use a (hash) function to map key to index of the table (array)
  - Element x is stored in T[h(x.key)]
  - hash function: int hash (Key k) // return value 0…m-1



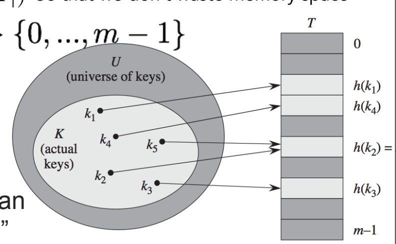
Collision: when two different keys are mapped to same index.

Can collision be avoided?

Is it possible to design a hash function that is one-to-one?
Hint: domain and condomain of hash()?

Hashing: unavoidable collision

- a large universe set **U**
- A set K of actually occurred keys, |K| << |U| (much much smaller)
- Table T of size m, $m = \Theta(|K|)$ So that we don't waste memory space
- A **hash function**: $h : U \rightarrow \{0, \ldots, m-1\}$
- Given |U| > |m|, hash function is many-to-one
  - by pigeonhole theorem
  - Collisions cannot be avoided but its chances can be reduced using a "good" hash function



HashTable Operations

- **If there is no collision:**
  - **Insert**
    - Table[h("john")]=Element("John", 25000)
  - **Delete**
    - Table[h("john")]=NULL
  - **Search**
    - return Table[h("john")]
- All constant time O(1)

key
Element value

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

Hash Function

- A **hash function**: $h : U \to \{0, ..., m-1\}$. Given an element x, x is stored in T[h(x.key)]
- Good hash function:
  - fast to compute
  - Ideally, map any key equally likely to any of the slots, independent of other keys
- Hash Function:
  - first stage: map non-integer key to integer
  - second stage: map integer to [0…m-1]

First stage: any type to integer

- Any basic type is represented in binary
- Composite type which is made up of basic type
  - a character string (each char is coded as an int by ASCII code), e.g.,"pt"
    - add all chars up, 'p'+'t'=112+116=228
    - radix notation: 'p'*128+'t'=14452
      - treat "pt" as base 128 number…
  - a point type: (x,y) an ordered pair of int
    - x+y
    - ax+by // pick some non-zero constants a, b
    - …
  - IP address:four integers in range of 0…255
    - add them up
    - radix notation: $150*256^3+108*256^2+68*256+26$

Hash Function: second stage

- **Division method**: divide integer by m (size of hash table) and take remainder
  - h(key) = key mod m
- if key's value are randomly uniformly distributed all integer values, the above hash function is uniform
- But often times data are not randomly distributed,
  - What if m=100, all keys have same last two digits?
  - Similarly, if $m=2^p$, then result is simply the lowest-ordre p bits
- Rule of thumbs: choose m to be a prime not too close to exact powers of 2

Hash Function: second stage

- **Multiplication method**: pick a constant A in the range of (0,1),

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

  - take fraction part of kA, and multiply with m
  - e.g., m=10000,$A = \sqrt{5} - 1/2 = 0.618033...$ h(123456)=41.
- Advantage: m could be exact power of 2…

Multiplication Method

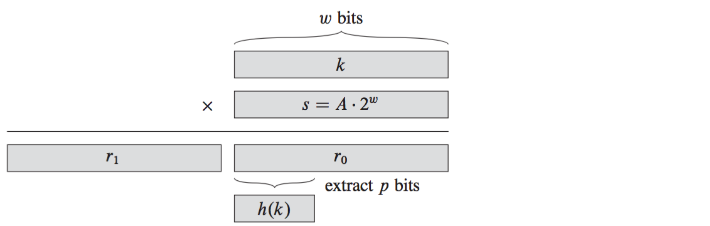


Figure 11.4 The multiplication method of hashing. The w-bit representation of the key k is multiplied by the w-bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w-bit half of the product form the desired hash value $h(k)$.

$$m = 2^p$$

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

Exercise

- Write a hash function that maps string type to a hash table of size 250
  - First stage: using radix notation
    - "Hello!" => <u>'H'*128^5+'e'*128^4+…+'!'</u>
  - Second stage: x
    - x mod 250
- How do you implement it efficiently?
  - Recall modular arithmetic theorem?
  - (x+y) mod n = ((x mod n)+(y mod n)) mod n
  - (x * y) mod n = ((x mod n)*(y mod n)) mod n
  - (x^e) mod n = (x mod n)^e mod n

Exercise

- Write a hash function that maps a point type as below to a hash table of size 100

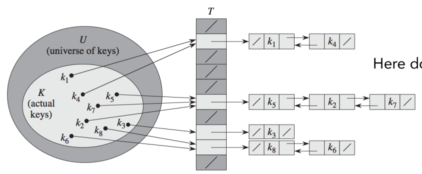
```
class point{
    int x, y;
}
```


Collision Resolution

- Recall that h(.) is not one-to-one, so it maps multiple keys to same slot:
  - for distinct k1, k2, h(k1)=h(k2) => collision
- Two different ways to resolve collision
  - Chaining: store colliding keys in a linked list (bucket) at the hash table slot
    - dynamic memory allocation, storing pointers (overhead)
  - Open addressing: if slot is taken, try another, and another (a probing sequence)
    - clustering problem.

Chaining

- Chaining: store colliding elements in a linked list at the same hash table slot
  - if all keys are hashed to same slot, hash table degenerates to a linked list.



Here doubly-linked list is used

Figure 11.3 Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_7) = h(k_2)$. The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

  - C++: NodePtr T[m];
  - STL: vector<list<HashedObject>> T;

Chaining: operations

- Insert (T,x):
  - insert x at the head of T[h(x.key)]
  - Running time (worst and best case): O(1)
- Search (T,k)
  - search for an element with key x in list T[h(k)]
- Delete (T,x)
  - Delete x from the list T[h(x.key)]
- Running time of search and delete: proportional to length of list stored in h(x.key)

Chaining: analysis

- Consider a hash table T with m slots stores n elements.
  - load factor $\alpha = n/m$
- If any given element is equally likely to hash into any of the m slots, independently of where any other element is hashed to, then average length of lists is α
  - search and delete takes $\Theta(1+\alpha)$
- If all keys are hashed to same slot, hash table degenerates to a linked list
  - search and delete takes $\Theta(n)$

Collision Resolution

- Open addressing: store colliding elements elsewhere in the table
  - Advantage: no need for dynamic allocation, no need to store pointers
- When inserting:
  - examine (probe) a sequence of positions in hash table until find empty slot
  - e.g., linear probing: if T[h(x.key)] is taken, try slots: h(x.key)+1, h(x.key+2), …
- When searching/deleting:
  - examine (probe) a sequence of positions in hash table until find element

Open Addressing

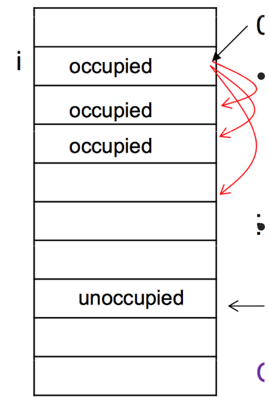
- Hash function: extended to probe sequence (m functions):

$$h_i(x), i = 0, 1, ..., m-1$$

$$h_i(x) \neq h_j(x), \text{ for } i \neq j$$

- **insert** element with key x: if $h_0(x)$ is taken, try $h_1(x)$, and then $h_2(x)$, until find an empty/deleted slot
- **Search** for key x: if element at $h_0(x)$ is not a match, try $h_1(x)$, and then $h_2(x)$, ..until find matching element, or reach an empty slot
- **Delete** key x: mark its slot as DELETED

Linear Probing



- Probing sequence
  - $h_i(x)=(h(x)+i)$ mod m
- probe sequence: h(x),h(x)+1, h(x)+2, …
  - Continue until an empty slot is found
- Problem: primary clustering
  - if there are multiple keys mapped to a slot, the slots after it tends to be occupied
  - Reason: all keys using same probing: +1, +2, …

Quadratic Probing

$$h_i(x) = (h(x) + c_1 i + c_2 i^2) \bmod m$$

- probe sequence:
  - $h_0(x)=h(x)$ mod m
  - $h_1(x)=(h(x)+c_1+c_2)$ mod m
  - $h_2(x)=(h(x)+2c_1+4c_2)$ mod m
  - …
- Problem:
  - secondary clustering
  - choose c_1,c_2,m carefully so that all slots are probed

Double Hashing

- Use two functions f_1,f_2:

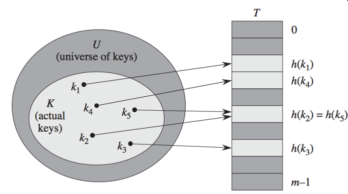
$$h_i(x) = (f_1(x) + i \cdot f_2(x)) \bmod m$$

- Probe sequence:
  - $h_0(x)=f_1(x)$ mod m,
  - $h_1(x)=(f_1(x)+f_2(x))$ mod m
  - $h_2(x)=(f_1(x)+2f_2(x))$ mod m,…
- $f_2(x)$ and m must be relatively prime for entire hash table to be searched/used
  - Two integers a, b are relatively prime with each other if their greatest common divisor is 1
  - e.g., $m=2^k$, $f_2(x)$ be odd
  - or, m be prime, $f_2(x)$<m

Exercises

- Hash function, Chaining, Open addressing
- Implementing HashTable
- Using C++ STL containers (implemented using hashtable)
  - unordered_set<int> // a set of int
  - unordered_map<string,int> lookup; //key, value
  - unordered_multiset
  - You can specify your own hash function
  - In contrast, set, map, multimap are implemented using binary search tree (keys are ordered)
  - All are associative container: where elements are referenced by key, not by position/index
    - e.g., lookup["john"]=100;

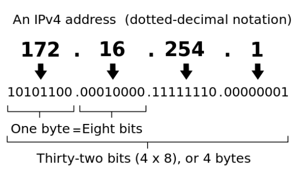
Design Hash Function



- Goal: reduce collision by spread the hash values uniformly to 0…m-1
  - so that for any key, it's equally likely to be hashed to 0, 1, …m-1
- We know the U, the set of possible values that keys can take
- But sometimes we don't know K beforehand…

Case studies

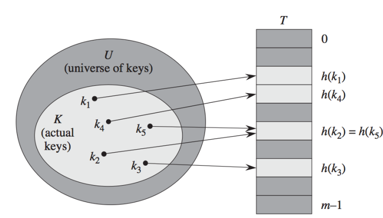
- A web server: maintains all active clients' info, using IP addr. as key



  - key is 32 bits long int, or $x_1.x_2.x_3.x_4$ (each 8 bits long, between 0 and 255)
- Let's try to use hash table to organize the data!
- Suppose that we expect about 250 active clients…
  - So we use a table of length 250 (m=250)

Hash function

- A hash function h maps IP addr to positions in the table
  - Each position of table is in fact a bucket (a linked list that contains all IP addresses that are mapped to it)
  - (i.e., chaining is used)



Design of Hash Function

- One possible hash function would map an IP address to the 8-bit number that is its last segment:
  - h(x1.x2.x3.x4) = x4 mod m
  - e.g., h(128.32.168.80) = 80 mod 250 = 250
- But is this a good hash function?
  - Not if the last segment of an IP address tends to be a small number; then low-numbered buckets would be crowded.
- Taking first segment of IP address also invites disaster, e.g., if most of our customers come from a certain area.

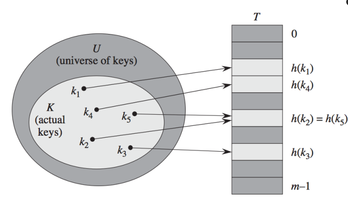
How to choose a hash function?

- There is nothing inherently wrong with these two functions.
- If our IP addr. were uniformly drawn from all 2^{32} possibilities, then these functions would behave well.
  - … the last segment would be equally likely to be any value from 0 to 255, so the table is balanced…
- The problem is we have no guarantee that the probability of seeing all IP addresses is uniform.
  - these are dynamic and changing over time.

How to choose a hash function?

- In most application:
  - fixed U, but the set of data K (i.e., IP addrs) are not necessarily uniformly randomly drawn from U
- There is no single hash function that behaves well on all possible sets of data.
- Given **any** hash function maps $|U|=2^{32}$ IP addrs to m=250 slots
  - there exists a collection of at least $2^{32}/250=2^{24}\approx 16{,}000{,}000$ IP addr that are mapped to same slot (or collide).
  - if data set K all come from this collection, hash table becomes linked list!

In General…



- If $|U| \geq (N-1)m+1$, then for any hash function h, there exists a set of N keys in U, such that all keys are hashed to same slot
  - Proof.(General pigeon-hole principle) if every slot has at most N-1 keys mapped to it under h, then there are at most (n-1)m elements in U. But we know |U| is larger than this, so …
- Implication: no matter how careful you choose a hash function, there is always some input (S) that leads to a linear insertion/deletion/search time

Solution: Universal Hashing

- For any **fixed** hash function, h(.), there exists a set of n keys, such that all keys are hashed to same slot
- Solution: randomly select a hash function from a carefully designed class of hash functions
  - For any input, we might choose a bad hash function on a run, and good hash function on another run…
  - averaged on different runs, performance is good

A family of hash functions

- Let us make the table size to be m = 257, a prime number!
- Every IP address x as a quadruple x = (x_1, x_2, x_3, x_4) of integers (all less than m).
- Fix any four numbers (less than 257), e.g., 87, 23, 125, and 4, we can define a function h() as follows:
- $h(x_1, x_2, x_3, x_4) = (81x_1 + 23x_2 + 125x_3 + 4x_4) \bmod 257$
- In general, for any four coefficients a_1,...,a_4 ∈{0,1,…, n−1}write a = (a_1, a_2, a_3, a_4), and define h_a as follows:

$$h_a(x_1, x_2, x_3, x_4) = (a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4) \bmod 257$$

Universal hash

Consider any pair of distinct IP addresses x = (x_1,...,x_4) and y = (y_1,...,y_4). If the coefficients a = (a_1, . . . , a_4) are chosen uniformly at random from {0,1,..., m− 1}, then

$$\Pr\left[h_a(x_1,\ldots,x_4) = h_a(y_1,\ldots,y_4)\right] = \frac{1}{m}.$$

- Proof omitted.
- Implication: given any pair of diff keys, the randomly selected hash function maps them to same slot with prob. 1/m.
- For a set S of data, the average/expected chain length is |S|/m=n/m=α
- => Very good average performance

A class of universal hash

Let

$$H = \{h_a | a \in \{0, 1, ..., m-1\}\}$$

The above set of hash functions is universal: For any two distinct data items x and y, exactly 1/m of all the hash functions in H map x and y to the slot, where n is the number of slots.

Two-level hash table

- Perfect hashing: if we fix the set S, can we find a hash function h so that all lookups are constant time?
- Use universal hash functions with 2-level scheme
  1. hash into a table of size m using universal hashing (some collision unless really lucky)
  2. rehash each slot, here we pick a random h, and try it out, if collision, try another one, …

Note: Cryptographic hash function

- It is a mathematical algorithm
  - maps data of arbitrary size to a bit string of a fixed size (a hash function)
  - designed to be **a one-way function**, that is, a function which is infeasible to invert.
  - only way to recreate input data from an ideal cryptographic hash function's output is to attempt a brute-force search of possible inputs to see if they produce a match, or use a "rainbow table" of matched hashes.

Properties of crypt. hash function

- Ideally,
  - it is deterministic so the same message always results in the same hash
  - it is quick to compute the hash value for any given message
  - it is infeasible to generate a message from its hash value except by trying all possible messages
  - a small change to a message should change the hash value so extensively that the new hash value appears uncorrelated with the old hash value
  - it is infeasible to find two different messages with the same hash value

Cryp. hash functions

- Application of crypt. hash function:
  - ensure integrity of everything from digital certificates for HTTPS websites, to managing commits in code repositories, and protecting users against forged documents.
- Recently, Google announced a public collision in the SHA-1 algorithm
  - with enough computing power — roughly 110 years of computing from a single GPU — you can produce a collision, effectively breaking the algorithm.
  - Two PDF files were shown to be hashed to same hash
  - Allow malicious parties to tamper with Web contents…