Algorithms and Data Structures (II)

Gabriel Istrate

March 10, 2020

Dictionary

- A dictionary is an abstract data structure that represents a set of elements (or keys)
    - a dynamic set

Dictionary

- A dictionary is an abstract data structure that represents a set of elements (or keys)
  - a dynamic set

- Interface (generic interface)
  - $\text{Insert}(D, k)$ adds a key k to the dictionary D
  - $\text{Delete}(D, k)$ removes key k from D
  - $\text{Search}(D, k)$ tells whether D contains a key k

Dictionary

- A dictionary is an abstract data structure that represents a set of elements (or keys)
  - a dynamic set
- Interface (generic interface)
  - $\mathrm{Insert}(D, k)$ adds a key k to the dictionary D
  - $\mathrm{Delete}(D, k)$ removes key k from D
  - $\mathrm{Search}(D, k)$ tells whether D contains a key k
- Implementation
  - many (concrete) data structures

Dictionary

- A dictionary is an abstract data structure that represents a set of elements (or keys)
  - a dynamic set
- Interface (generic interface)
  - $\mathrm{Insert}(D, k)$ adds a key k to the dictionary D
  - $\mathrm{Delete}(D, k)$ removes key k from D
  - $\mathrm{Search}(D, k)$ tells whether D contains a key k
- Implementation
  - many (concrete) data structures
  - we'll see: hash tables

Direct-Address Table

- A direct-address table implements a dictionary

Direct-Address Table

- A direct-address table implements a dictionary
- The universe of keys is $U = \{1, 2, \ldots, M\}$

Direct-Address Table

- A direct-address table implements a dictionary
- The universe of keys is $U = \{1, 2, \ldots, M\}$
- Implementation
  - an array T of size M
  - we store keys directly in T !

Direct-Address Table

- A direct-address table implements a dictionary
- The universe of keys is $U = \{1, 2, \ldots, M\}$
- Implementation
  - an array T of size M
  - we store keys directly in T !

```
Direct-Address-Insert(T, k)
1   T[k] = true
```

```
Direct-Address-Delete(T, k)
1   T[k] = false
```

```
Direct-Address-Search(T, k)
1   return T[k]
```

Direct-Address Table (2)

- Complexity

Direct-Address Table (2)

- Complexity

All direct-address table operations are O(1)✓

Direct-Address Table (2)

- Complexity

  All direct-address table operations are O(1)✓

  So why isn't every set implemented with a direct-address table?

Direct-Address Table (2)

- Complexity

  All direct-address table operations are $O(1)$✓

  So why isn't every set implemented with a direct-address table?

- Space complexity is $\Theta(|U|)$×
  - $|U|$ is typically a very large number—U is the universe of keys!
  - the represented set is typically much smaller than $|U|$
    - i.e., a direct-address table usually wastes a lot of space

Direct-Address Table (2)

- Complexity

  All direct-address table operations are O(1)✓

  So why isn't every set implemented with a direct-address table?

- Space complexity is $\Theta(|U|)\times$
  - $|U|$ is typically a very large number—U is the universe of keys!
  - the represented set is typically much smaller than $|U|$
    - i.e., a direct-address table usually wastes a lot of space

- Want: the benefits of a direct-address table but with a table of reasonable size.

Direct Access Tables: Scorecard

Algorithm	Complexity
INSERT	O(1)✓
DELETE	O(1)✓
SEARCH	O(1)✓
MEMORY:	θ(M)×

Hash Tables

- Idea
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a hash function

$$h : U \to \{1, \ldots, |T|\}$$

  - $h(k)$ easy ($O(1)$) to compute given k

Hash Tables

- Idea
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a hash function

$$h : U \to \{1, \ldots, |T|\}$$

  - $h(k)$ easy ($O(1)$) to compute given k

```
Hash-Insert(T, k)
1  T[h(k)] = true
```

```
Hash-Delete(T, k)
1  T[h(k)] = false
```

```
Hash-Search(T, k)
1  return T[h(k)]
```

Hash Tables

- Idea
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a hash function

$$h : U \to \{1, \ldots, |T|\}$$

  - $h(k)$ easy ($O(1)$) to compute given k

```
Hash-Insert(T, k)
1   T[h(k)] = true
```

```
Hash-Delete(T, k)
1   T[h(k)] = false
```

```
Hash-Search(T, k)
1   return T[h(k)]
```

Are these algorithms always correct?

Hash Tables

- Idea
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a hash function

$$h : U \to \{1, \ldots, |T|\}$$

  - h(k) easy (O(1)) to compute given k

```
Hash-Insert(T, k)
1   T[h(k)] = true
```

```
Hash-Delete(T, k)
1   T[h(k)] = false
```

```
Hash-Search(T, k)
1   return T[h(k)]
```

Are these algorithms always correct? No!

Hash Tables

- Idea
  - use a table T with $|T| \ll |U|$
  - map each key $k \in U$ to a position in T, using a hash function

$$h : U \to \{1, \ldots, |T|\}$$

  - $h(k)$ easy ($O(1)$) to compute given k

```
Hash-Insert(T, k)
1   T[h(k)] = true
```

```
Hash-Delete(T, k)
1   T[h(k)] = false
```

```
Hash-Search(T, k)
1   return T[h(k)]
```

Are these algorithms always correct? No!

What if two distinct keys $k_1 \neq k_2$ collide? (I.e. $h(k_1) = h(k_2)$)

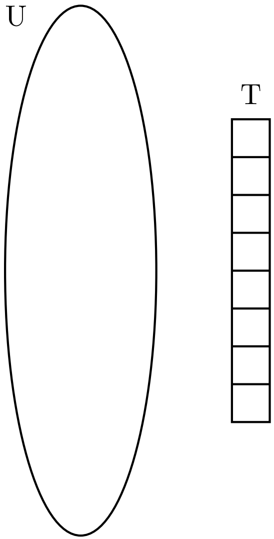
Hash tables

- Work well "on the average"
- Analogy: throw T balls at random into N bins.
- If T << N (in fact $T = o(\sqrt{N})$) then with high-probability no two balls land in the same bin.
- On the average: T/N balls in each bin.
- Want our hash-function to be "random-like": elements of U "thrown out uniformly" by h onto elements of T.

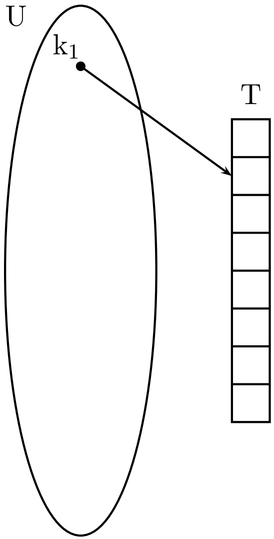
Hashing With Chaining

- Store all objects that map to the same bucket in a linked list.
- ”Hope” that hash function is ”uniform enough”, linked lists are not too large, set operations are efficient.

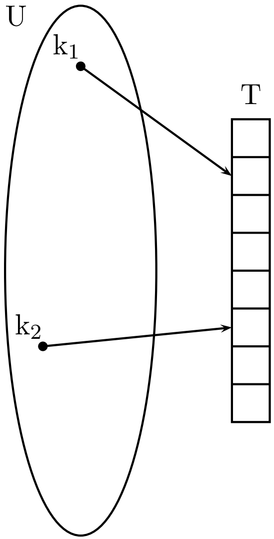
Hash Table: Chaining



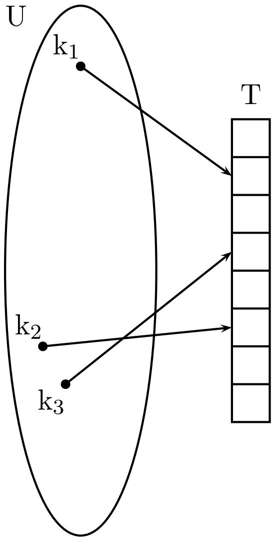
Hash Table: Chaining



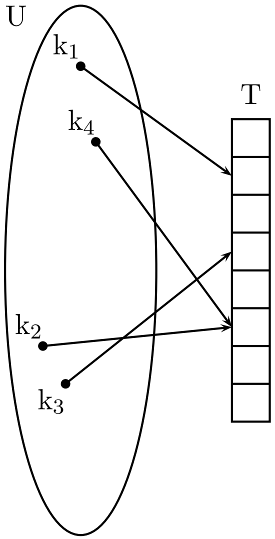
Hash Table: Chaining



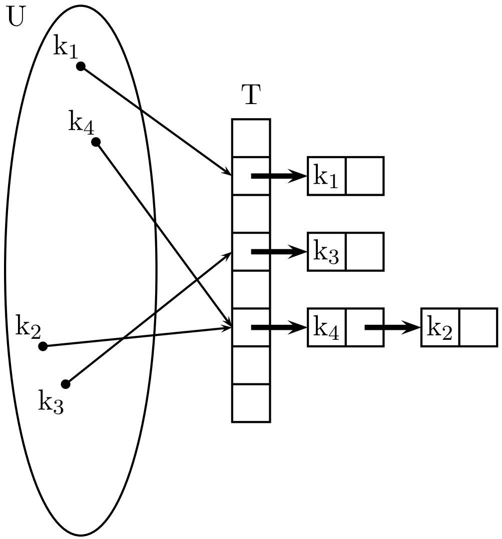
Hash Table: Chaining



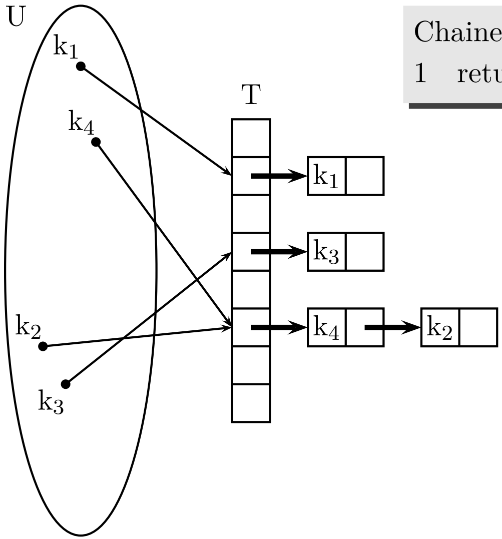
Hash Table: Chaining



Hash Table: Chaining

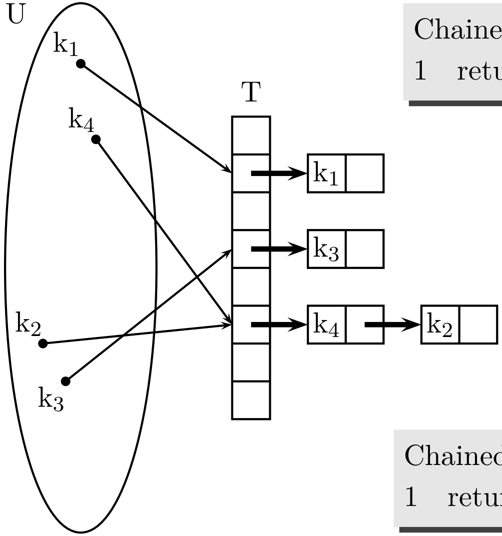


Hash Table: Chaining



```
Chained-Hash-Insert(T, k)
1   return List-Insert(T[h(k)], k)
```

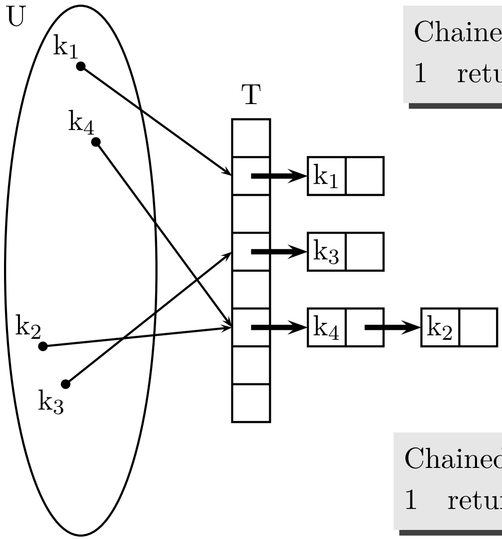
Hash Table: Chaining



```
Chained-Hash-Insert(T, k)
1   return List-Insert(T[h(k)], k)
```

```
Chained-Hash-Search(T, k)
1   return List-Search(T[h(k)], k)
```

Hash Table: Chaining



```
Chained-Hash-Insert(T, k)
1   return List-Insert(T[h(k)], k)
```

load factor
$\alpha = \frac{n}{|T|}$

```
Chained-Hash-Search(T, k)
1   return List-Search(T[h(k)], k)
```

Hashing With Chaining: Analysis

- We assume uniform hashing for our hash function $h : U \to \{1 \ldots |T|\}$ (where $|T| = T.\text{length}$)

Hashing With Chaining: Analysis

- We assume uniform hashing for our hash function $h : U \to \{1 \ldots |T|\}$ (where $|T| = T.\mathrm{length}$)

$$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \ldots |T|\}$$

(The formalism is actually a bit more complicated.)

Hashing With Chaining: Analysis

- We assume uniform hashing for our hash function $h : U \to \{1 \dots |T|\}$ (where $|T| = T.\text{length}$)

$$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \dots |T|\}$$

  (The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length n_i of the linked list at position i is

$$E[n_i] = \frac{n}{|T|} = \alpha$$

Hashing With Chaining: Analysis

- We assume uniform hashing for our hash function $\mathrm{h} : \mathrm{U} \to \{1 \ldots |\mathrm{T}|\}$ (where $|\mathrm{T}| = \mathrm{T.length}$)

  $$\Pr[\mathrm{h}(\mathrm{k}) = \mathrm{i}] = \frac{1}{|\mathrm{T}|} \quad \text{for all } \mathrm{i} \in \{1 \ldots |\mathrm{T}|\}$$

  (The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length $\mathrm{n_i}$ of the linked list at position i is

  $$\mathrm{E}[\mathrm{n_i}] = \frac{\mathrm{n}}{|\mathrm{T}|} = \alpha$$

- We further assume that h(k) can be computed in O(1) time

Hashing With Chaining: Analysis

- We assume uniform hashing for our hash function $h : U \to \{1 \ldots |T|\}$ (where $|T| = T.\text{length}$)

$$\Pr[h(k) = i] = \frac{1}{|T|} \quad \text{for all } i \in \{1 \ldots |T|\}$$

  (The formalism is actually a bit more complicated.)

- So, given n distinct keys, the expected length n_i of the linked list at position i is

$$E[n_i] = \frac{n}{|T|} = \alpha$$

- We further assume that $h(k)$ can be computed in $O(1)$ time

- Therefore, the complexity of Chained-Hash-Search is

$$\boxed{\Theta(1 + \alpha)}$$

Hashing with Open Addressing

- Alternative to chaining: instead of using linked lists,store all the elements in the table
  - this implies $\alpha \leq 1$

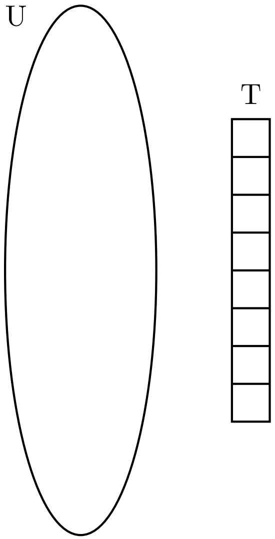
Hashing with Open Addressing

- Alternative to chaining: instead of using linked lists,store all the elements in the table
  - this implies $\alpha \leq 1$
- When a collision occurs, simply find another free cell in T

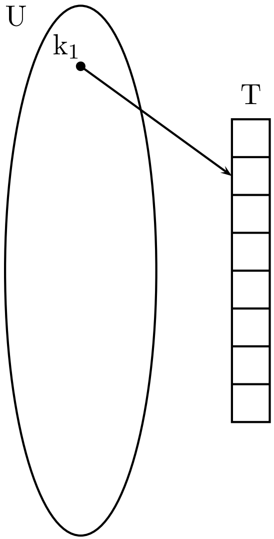
Hashing with Open Addressing

- Alternative to chaining: instead of using linked lists,store all the elements in the table
    - this implies $\alpha \leq 1$
- When a collision occurs, simply find another free cell in T
- A sequential "probing" method may not be optimal
    - can you imagine why?

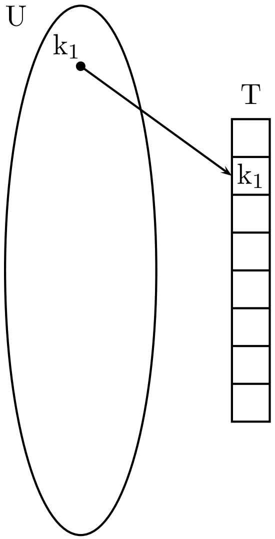
Open-Address Hash Table



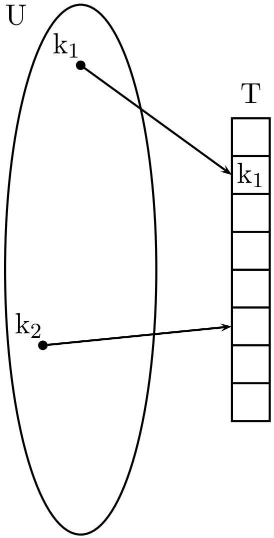
Open-Address Hash Table



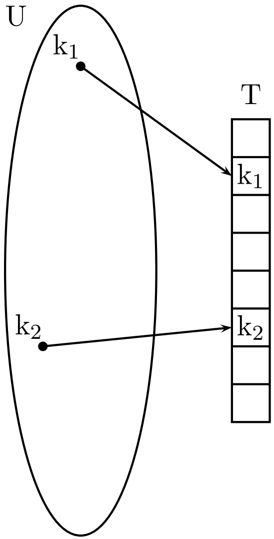
Open-Address Hash Table



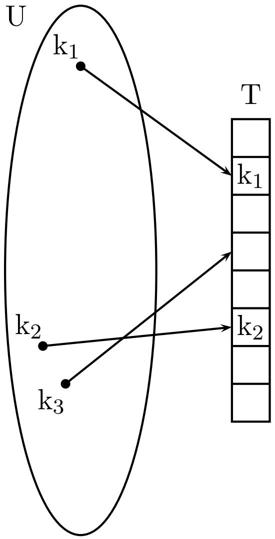
Open-Address Hash Table



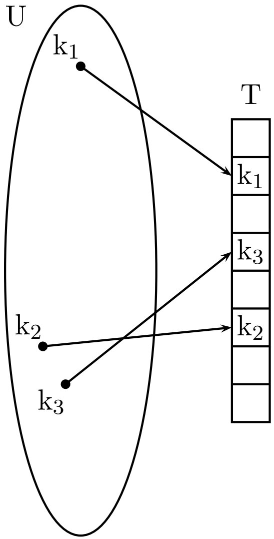
Open-Address Hash Table



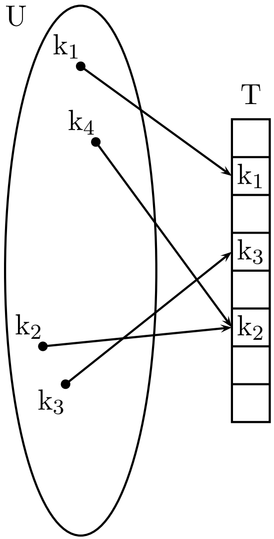
Open-Address Hash Table



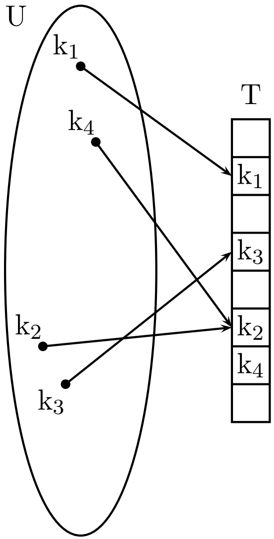
Open-Address Hash Table



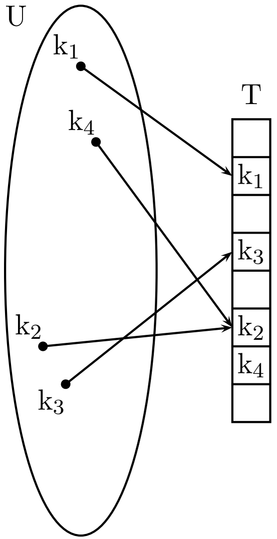
Open-Address Hash Table



Open-Address Hash Table



Open-Address Hash Table



```
Hash-Insert(T, k)
1   j = h(k)
2   for i = 1 to T.length
3        if T[j] == nil
4             T[j] = k
5             return j
6        elseif j < T.length
7             j = j + 1
8        else j = 1
9   error "overflow"
```

Open-Addressing (3)

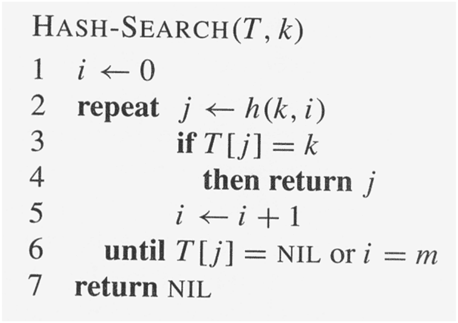
```
Hash-Insert(T, k)
for i = 1 to T.length
j = h(k, i)
    if T[j] == nil
        T[j] = k
        return j
error "overflow"
```

Open-Addressing (3)

```
Hash-Insert(T, k)
1   for i = 1 to T.length
2   j = h(k, i)
3           if T[j] == nil
4                   T[j] = k
5                   return j
6   error “overflow”
```

- Notice that $h(k, \cdot)$ must be a permutation
  - i.e., $h(k,1), h(k,2), \ldots, h(k,|T|)$ must cover the entire table T

Procedure HASH-SEARCH



```
HASH-SEARCH($T, k$)
$i \leftarrow 0$
repeat $j \leftarrow h(k, i)$
       if $T[j] = k$
          then return $j$
       $i \leftarrow i + 1$
   until $T[j] =$ NIL or $i = m$
return NIL
```

Open-address hashing

- Deletion: difficult. Marking NIL does not work.
- Doing so might make it impossible to retrieve any key during whose insertion probed slot i and found it occupied.
- One solution: DELETED instead of NIL. Problem: search time no longer dependent on load factor.
- Techniques for probing: linear probing, quadratic probing and double hashing.
- Linear probing: given auxiliary hash function $h' : U \to \{0, \ldots, m-1\}$, use hash function

$$h(k, i) = (h'(k) + i) \bmod m.$$

- Easy to implement but suffers from problem called primary clustering.
- Long runs of occupied slots build up, increasing average search time.

Open-address hashing

- Quadratic probing

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

- Works much better than linear probing, but to make use of full hash table the values of c_1, c_2, m are constrained.
- Suffers from secondary clustering: if two keys have the same initial probe position then their probe sequences are the same.

Open-address hashing

- Double hashing

$$h(k, i) = (h_1(k) + ih_2(k)) \bmod m.$$

- Among the best methods for open addressing.
- $h_2(k)$ must be relative prime to m. One solution is m a power of two and $h_2(k)$ odd.
- Another one: m prime, $h_2(k) < m$.
- Given an open address hash table with load factor $\alpha = n/m < 1$ the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

Double hashing

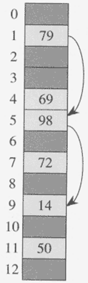


Figure 11.5 Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

Good hash functions

Caution

- The area of designing good hash function huge.
- Theoreticians and practitioners as well.
- Many hash functions good for specific goal.
- Appearing next does not mean you should blindly use them !
- Drozdek: discusses some more "practical" examples.
- Here: we follow CORMEN, concentrate on general ideas.

Requirements

- A good hash function satisfies (approximately) the assumption of uniform hashing.
- Unfortunately, usually we don't know probability distribution of the keys, and keys might not be drawn independently.

Good hash functions

- Good case: if items are random real numbers k uniformly distributed in $[0, 1)$, $h(k) = \lfloor km$ satisfies simple uniform hashing conditions.
- Most hash functions assume universe of keys are the natural numbers.
- E.g. character string = integer in base 128 notation.
- Identifier pt. ASCII $p = 112$, $t = 116$, becomes $112 \cdot 128 + 116 = 14452$.
- Division method: $h(k) = k \bmod m$. Avoid some values of m,e.g. powers of two. Indeed, if $m = 2^p$ then $h(k) =$ the p lowest bits of k. Unless we know that p lowest bits of keys are uniform not a good idea.
- Prime not too close to an exact power of two = often a good choice.

Folding

- Input: broken into pieces.
- Combined in some way.

Example

- SSN (American CNP): 123456789.
- Divide into three parts: 123, 456, 789.
- Add these: 1368.
- Reduced modulo table-size (1000): 368

Good hash functions: Multiplication method

- Multiplication method: Two stage procedure
- First, multiply key by constant A in range $0 < A < 1$, extract fractional part.
- Then multiply by m, extract floor.
- $h(k) = \lfloor m(kA \bmod 1) \rfloor$.
- Value of m not critical. $m = 2^p$.
- Easy implementation. Restrict $A = s/2^w$, w=machine word size.
- Better with some values of A than other. Knuth suggests $A \sim (\sqrt{5} - 1)/2$ will work well.

Multiplication-method of hashing

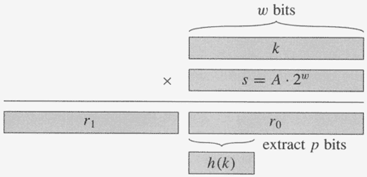
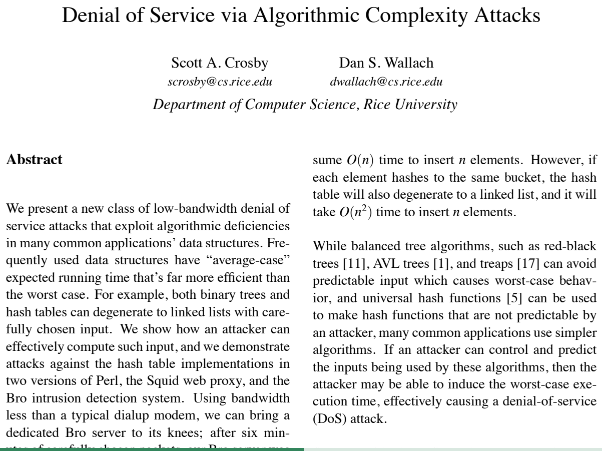


Figure 11.4 The multiplication method of hashing. The w-bit representation of the key k is multiplied by the w-bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w-bit half of the product form the desired hash value $h(k)$.

Universal hashing

- Any fixed hash function vulnerable to worst case behavior:
- if "adversary" chooses n keys that all hash to the same slot, this yields average search time of θ(n).
- Practical example: Crosby and Wallach (USENIX'03) have shown that one can slow down to a halt systems by attacking implementations of hash tables in Perl, squid web proxy, Bro intrusion detection.
- Cause: hashing mechanism known (due to, e.g. publicly available implementation).
- Solution: choose hash function randomly, independent of the keys that are going to be stored.

Attack when hashing mechanism known



Denial of Service via Algorithmic Complexity Attacks

Scott A. Crosby *scrosby@cs.rice.edu*
Dan S. Wallach *dwallach@cs.rice.edu*
Department of Computer Science, Rice University

Abstract

We present a new class of low-bandwidth denial of service attacks that exploit algorithmic deficiencies in many common applications' data structures. Frequently used data structures have "average-case" expected running time that's far more efficient than the worst case. For example, both binary trees and hash tables can degenerate to linked lists with carefully chosen input. We show how an attacker can effectively compute such input, and we demonstrate attacks against the hash table implementations in two versions of Perl, the Squid web proxy, and the Bro intrusion detection system. Using bandwidth less than a typical dialup modem, we can bring a dedicated Bro server to its knees; after six min-

sume $O(n)$ time to insert n elements. However, if each element hashes to the same bucket, the hash table will also degenerate to a linked list, and it will take $O(n^2)$ time to insert n elements.

While balanced tree algorithms, such as red-black trees [11], AVL trees [1], and treaps [17] can avoid predictable input which causes worst-case behavior, and universal hash functions [5] can be used to make hash functions that are not predictable by an attacker, many common applications use simpler algorithms. If an attacker can control and predict the inputs being used by these algorithms, then the attacker may be able to induce the worst-case execution time, effectively causing a denial-of-service (DoS) attack.

Universal hashing

- $\mathcal{H}$ finite collection of hash functions that map universe U into $\{0, 1, \ldots, m - 1\}$.
- Such a collection is called universal if for every keys $k \neq l \in U$, the number of hash functions $h \in \mathcal{H}$ for which $h(k) = h(l)$ is at most $|\mathcal{H}/m|$.
- Suppose a hash function h is chosen from a universal collection of hash functions, and is used to hash n keys into a table T of size m (using chaining).
- If key is not in the table expected length of the list that k hashes to is at most α.
- If key is not in the table expected length of the list that k hashes to is at most $1 + \alpha$.

A universal class of hash functions

- Due to Carter and Wegman.
- $a \equiv b \pmod{p}$ if $p|(a-b)$.
- Z_p: integers modulo p. p prime.
- How do we choose p ? So that all keys are in the range 0 to $p-1$.
- m: number of slots in the hash table.
- $a \in Z_p^*$, $b \in Z_p$.
- $h_{a,b}(k) = ((ak+b) (\bmod p)) \pmod{m}$.
- $\mathcal{H}_{p,m} = \{h_{a,b} : a \in Z_p^*, b \in Z_p\}$.
- Other applications of this set of hash functions: pseudo-random generators.

Perfect hashing

- Hashing can provide worst-case performance when the set of keys is static: once stored in the table, the set of keys never changes.
- Example: set of files on a DVD-R (finished).
- Perfect hashing: the worst-case number of accesses to perform a search is O(1).
- Idea: two-level hashing with universal hashing at each level.
- First level: the n keys are hashed into m slots using a hashing function chosen from a family of universal hash functions.
- Instead of chaining: Use (small) secondary table S_j with an associated hash function h_j.
- Choosing h_j carefully guarantees no collisions.

Perfect hashing with chaining

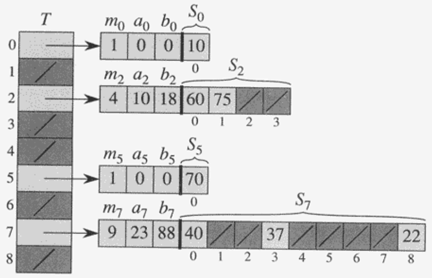


Figure 11.6 Using perfect hashing to store the set $K = \{10, 22, 37, 40, 60, 70, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is m_j, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 1$, key 75 is stored in slot 1 of secondary hash table S_2. There are no collisions in any of the secondary hash tables, and so searching takes constant time in the worst case.

Perfect hashing: design

- n_j = number of elements that hash to slot j.
- We let $m_j = |S_j| = n_j^2$.
- Idea: if $m = n^2$ and we store n keys in a table of size $m = n^2$ using a hash function randomly chosen from a set of universal hash function then the collision probability is at most 1/2.
- Find a good hash function using O(1) trials.
- Expected amount of memory O(n).
- Why this works: proof omitted (see Cormen if curious).

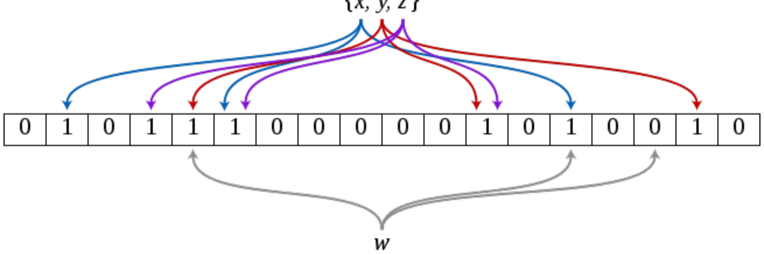
Hashing: there is more

- Cryptographic hash functions: hash functions with good security properties.
- Most well-known cryptographic hash function: md5 (Rabin). You probably have encountered it if you downloaded anything large from the web.
- (sha-1), sha-2, sha-3.
- U.S. Government standards.
- (Some) attacks on sha-1 (CWI Amsterdam, 2017)

SHA1("The quick brown fox jumps over the lazy dog") gives hexadecimal: 2fd4e1c67a2d28fced849ee1bb76e7391b93eb12 gives Base64 binary to ASCII text encoding: L9ThxnotKPzthJ7hu3bnORuT6xI=

Bloom filters

- Probabilistic data structure. Used to test membership of an element in a dataset.
- NO answer: correct
- YES anwer: possibly false positive.

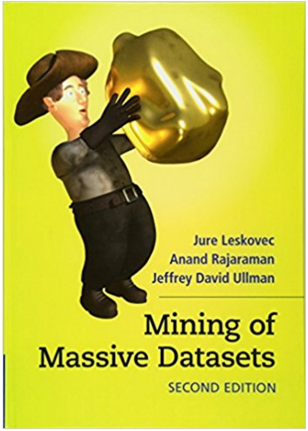


Hashing: where to go from here

- MapReduce model for grid computing: programming with hash functions.
- Data: key-value pairs.
- Map: applied in parallel to every pair (keyed by k1). Produces a list of pairs (keyed by k2) for each call.
- Mapreduce collects all pairs with the same k2 and groups them together.
- Reduce: applied in parallel to each group, which in turn produces a collection of values in the same domain.

Hashing: where to go from here

- Locality-sensitive hashing: reduces the dimensionality of high-dimensional data. LSH hashes input items so that similar items map to the same 'buckets' with high probability.



Hashing in programming languages

- Python: dictionaries.
- Hash tables in STL: some implementations (e.g. SGI). Most functionality provided by associative container map (implemented using red-black trees).
- However: C++-11: <u>two</u> implementations, std::unordered_map and std::unordered_set.