Hashing

Hash functions

Separate chaining

Linear probing

Double hashing

Reference: Chapter 14, Algorithms in Java, 3rd Edition, Robert Sedgewick.

Princeton University • COS 226 • Algorithms and Data Structures • Spring 2004 • Kevin Wayne • http://www.Princeton.EDU/~cos226

Optimize Judiciously

> "More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity." - William A. Wulf

> "We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil." - Donald E. Knuth

> "We follow two rules in the matter of optimization:
> Rule 1: Don't do it.
> Rule 2 (for experts only). Don't do it yet - that is, not until you have a perfectly clear and unoptimized solution."
> - M. A. Jackson

Reference: *Effective Java* by Joshua Bloch.

Hashing: Basic Plan.

Save items in a key-indexed table. Index is a function of the key.

Hash function. Method for computing table index from key.

Collision resolution strategy. Algorithm and data structure to handle two keys that hash to the same index.

Classic space-time tradeoff.

- No space limitation: trivial hash function with key as address.
- No time limitation: trivial collision resolution = sequential search.
- Limitations on both time and space: hashing (the real world)

Choosing a Good Hash Function

Goal: scramble the keys.

- Each table position equally likely for each key.

thoroughly researched problem

Ex: Social Security numbers.

- Bad: first three digits.
- Better: last three digits.

573 = California, 574 = Alaska

assigned in chronological order within a given geographic region

Ex: date of birth.

- Bad: first three digits of birth year.
- Better: birthday.

198 for all of you

less collisions even with only 366 possible values

Ex: phone numbers.

- Bad: first three digits.
- Better: last three digits.

Hash Function: String Keys

Strings hash functions.

- Java 1.1: calculation involving only 16 characters.
- Java 1.2: calculation involving all characters.

```
public int hashCode() {
   int hash = 0;
   for (int i = 0; i < length(); i++)
      hash = (31 * hash) + charAt(i);
   return hash;
}
```

String.java

```
s    = "call";
h    = s.hashCode();
hash = h % M;
```

7121 8191 3045982

Horner's method

- Equivalent to $h = 31^{N-1}s_0 + \ldots + 31^2 s_2 + 31 s_1 + s_{N-1}$.
- Can we use h % M as index for table of size M?

Work to hash a string of length W.

- W add, W multiply, 1 mod.
- Note: reference Java implementation caches `String` hash codes.

Collisions

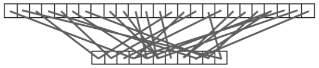
Collision = two keys hashing to same value.

- Essentially unavoidable.
- Birthday problem: how many people will have to enter a room until two have the same birthday? 23
- With M hash values, expect a collision after sqrt(π M/2) insertions.

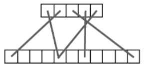
Conclusion: can't avoid collisions unless you have a ridiculous amount of memory.

Challenge: efficiently cope with collisions.

25 items, 11 table positions
~2 items per table position



5 items, 11 table positions
~ .5 items per table position



Collision Resolution.

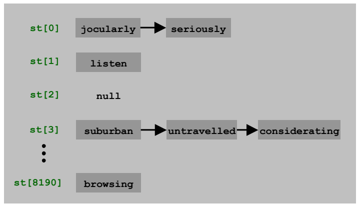
Two main approaches.

Separate chaining.

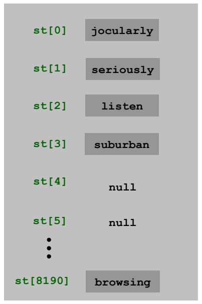
- M much smaller than N.
- ~N / M keys per table position.
- Put keys that collide in a list.
- Need to search lists.

Open addressing.

- M much larger than N.
- plenty of empty table slots.
- When a new key collides, find next empty slot and put it there.
- Complex collision patterns.



M = 8191, N = 15000

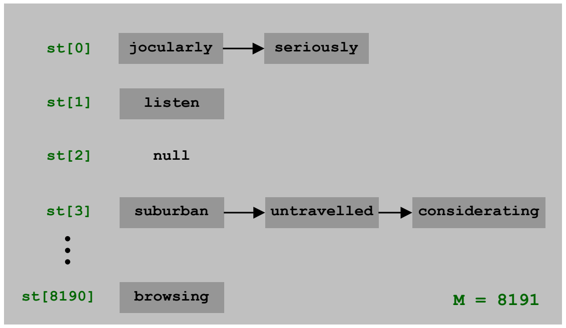


M = 30001, N = 15000

Separate Chaining

Separate chaining: array of M linked lists.

- Hash: map key to integer i between 0 and M-1.
- Insert: put at front of i^th chain. ⇦ constant time
- Search: only need to search i^th chain. ⇦ proportional to length of chain



key	hash
call	7121
me	3480
ishmael	5017
seriously	0
untravelled	3
suburban	3
. . .	.

Symbol Table: Hash Table Implementation

```
public class SymbolTable {
   private int M = 8191;
   private List[] st = new List[M];

   private class List { AS BEFORE }

   public static int hash(String s, int M) {
      return (s.hashCode() & 0x7fffffff) % M;
   }

   void put(String k, Object val) {
      int i = hash(k, M);
      st[i] = new List(k, val, st[i]);
   }

   Object get(String k) {
      int i = hash(k, M);
      for (List x = st[i]; x != null; x = x.next)
         if (k.equals(x.key)) return x.value;
      return null;
   }
}
```

number of chains (8191 is prime)

hex

between 0 and M-1

insert at front of i^{th} chain

exhaustively search i^{th} chain for key

Hash Table Implementation: Performance

Advantages: fast insertion, fast search.
Disadvantage: hash table has fixed size.

corrected by doubling the size of the array
and rehashing all of the key-value pairs

Hash tables improves ALL symbol table clients.

- Makes difference between practical solution and no solution.
- Ex: Moby Dick now takes a few seconds instead of hours.

```
% java DeDup < mobydick.txt
moby
dick
herman
melville
call
me
ishmael
some
years
ago
. . .
```

210,028 words
16,834 distinct

Separate Chaining Performance

Separate chaining performance.

- Search cost is proportional to length of chain.
- Trivial: average length = N / M.
- Worst case: all keys hash to same chain.

Theorem. Let $\alpha = N / M > 1$ be average length of list. For any $t > 1$, probability that list length $> t \alpha$ is exponentially small in t.

depends on hash map
being random map

Parameters.

- M too large $\Rightarrow$ too many empty chains.
- M too small $\Rightarrow$ chains too long.
- Typical choice: $\alpha = N / M \sim 10$ $\Rightarrow$ constant-time search/insert.

Symbol Table: Implementations Cost Summary

	Worst Case			Average Case		
Implementation	Search	Insert	Delete	Search	Insert	Delete
Sorted array	log N	N	N	log N	N / 2	N / 2
Unsorted list	N	1	1	N / 2	1	1
Hashing	N	1	N	1*	1*	1*

* assumes hash function is random

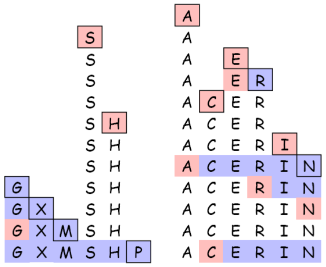
Linear Probing

Linear probing: array of size M. ⇐ typically twice as many slots as elements

- Hash: map key to integer i between 0 and M-1.
- Insert: put in slot i if free, if not try i+1, i+2, etc.
- Search: search slot i, if occupied but no match, try i+1, i+2, etc.

Cluster.

- Contiguous block of items.
- Search through cluster using elementary algorithm for arrays.



Linear Probing Performance

Linear probing performance.

- Insert and search cost depend on length of cluster.
- Trivial: average length of cluster = $\alpha = N / M$. ⇐ but elements more likely to hash to big clusters
- Worst case: all keys hash to same cluster.

Theorem (Knuth, 1962). Let $\alpha = N / M < 1$ be average length of list.

insert: $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$

search: $\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$

⇐ depends on hash map being random map

Parameters.

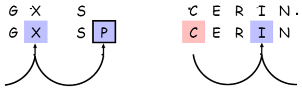
- M too large ⇒ too many empty array entries.
- M too small ⇒ clusters coalesce.
- Typical choice: M ~ 2N ⇒ constant-time search/insert.

Double Hashing

Double hashing: avoid clustering by using second hash to compute skip for search.

Hash: map key to integer i between 0 and M-1.
Second hash: map key to nonzero skip value. ⇐ best if relatively prime to M

Ex: 1 + (k mod 97).



Avoids clustering.

- Skip values give different search paths for keys that collide.

Double Hashing Performance

Linear probing performance.

- Insert and search cost depend on length of cluster.
- Trivial: average length of cluster = $\alpha = N / M$.
- Worst case: all keys hash to same cluster.

Theorem. Let $\alpha = N / M < 1$ be average length of list.

insert: $\frac{1}{1-\alpha}$

search: $\frac{1}{\alpha}\ln(1+\alpha)$

⇐ depends on hash map being random map

Parameters.

- M too large ⇒ too many empty array entries.
- M too small ⇒ clusters coalesce.
- Typical choice: M ~ 2N ⇒ constant-time search/insert.

Disadvantage: delete cumbersome to implement.

Hashing Tradeoffs

Separate chaining vs. linear probing/double hashing.

- Space for links vs. empty table slots.
- Small table + linked allocation vs. big coherent array.

Linear probing vs. double hashing.

		load factor α			
		50%	66%	75%	90%
linear probing	search	1.5	2.0	3.0	5.5
	insert	2.5	5.0	8.5	55.5
double hashing	search	1.4	1.6	1.8	2.6
	insert	1.5	2.0	3.0	5.5

Symbol Table: Java Libraries

Java has built-in libraries for symbol tables.

- `HashMap` = linear probing hash table implementation.

```
import java.util.HashMap;
public class HashMapDemo {
   public static void main(String[] args) {
      HashMap st = new HashMap();
      st.put("www.cs.princeton.edu", "128.112.136.11");
      st.put("www.princeton.edu",    "128.112.128.15");
      st.put("www.simpsons.com",     "209.052.165.60");
      System.out.println(st.get("www.cs.princeton.edu"));
   }
}
```

Duplicate policy.

- Java `HashMap` forbids two elements with the same key.
- Sedgewick implementations allow duplicate keys.

Implementing a HashMap Key

Java `HashMap` allows arbitrary objects as the key.

- Uses the `equals` and `hashCode` methods of the key object.
- Consistency: equal objects must have equal hash codes.
- Immutability: once you insert a key, don't change it a way that would change its `hashCode` or `equals`.
  - immutable in Java: `String`, `Integer`, `BigInteger`
  - mutable in Java: `Date`

> "Note: great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is changed in a manner that affects equals comparisons while the object is a key in the map. A special case of this prohibition is that it is not permissible for a map to contain itself as a key. "

Javadoc for Map interface

Implementing a HashMap Key

Phone numbers: (609) 867-5309.

area code exchange extension

```
public class PhoneNumber {
   private int area;   // area code (3 digits)
   private int exch;   // exchange  (3 digits)
   private int ext;    // extension (4 digits)

   // constructor, toString, but no mutators

   public boolean equals(Object x) {
      PhoneNumber a = this;
      PhoneNumber b = (PhoneNumber) x;
      return (a.area == b.area) &&
             (a.exch == b.exch) && (a.ext == b.ext);
   }

   public int hashCode() {
      return 10007 * (area + 1009 * exch) + ext;
   }
}
```

Frequency Symbol Table

Frequency symbol table.

- `fst.hit(key)` increment frequency count of given key.
- `fst.freq(key)` returns number of times given key occurs.

Applications.

- Web traffic analyzer: look up host to find number of hits.
- Browser: highlight visited links in purple.
- Chess: detect a repetition draw.
- Bayesian spam filter.

Implementation. Simple extension of a symbol table.

```
FrequencyTable fst = new FrequencyTable();
while (!StdIn.isEmpty() {
   String key = StdIn.readString();
   fst.hit(key);
   System.out.println(fst.freq(key));
}
```

Frequency Symbol Table

```
public class FrequencyTable {
   HashMap st = new HashMap();
   private class Entry {
      String name;
      int freq;
   }

   public void hit(String key) {
      Entry entry = (Entry) st.get(key);
      if (entry == null) {
         entry = new Entry();
         entry.name = key;
         st.put(key, entry);
      }
      entry.freq++;
   }

   public int freq(String key) {
      Entry entry = (Entry) st.get(key);
      if (entry == null) return 0;
      else return entry.freq;
   }
}
```

helper data type

create new entry if first occurrence of key

increment counter

return frequency

A Plan for Spam

Bayesian spam filter.

- Filter based on analysis of previous messages.
- User trains the filter by classifying messages as spam or ham.
- Parse messages into tokens (alphanumeric, dashes, ', $)

Build data structures.

- Hash table A of tokens and frequencies for spam.
- Hash table B of tokens and frequencies for ham.
- Hash table C of tokens with probability p that they appear in spam.

```
double h = 2.0 * ham.freq(word);
double s = 1.0 * spam.freq(word);
double p = (s/spams) / (h/hams + s/spams);
```

bias probabilities to avoid false positives

Reference: http://www.paulgraham.com/spam.html

A Plan for Spam

Identify incoming email as spam or ham.

- Find 15 most interesting tokens (difference from 0.5).
- Combine probabilities using Bayes law.

which data structure?

$$\frac{p_1 \times p_2 \times \cdots \times p_{15}}{(p_1 \times p_2 \times \cdots \times p_{15}) \;+\; ((1-p_1) \times (1-p_2) \times \cdots \times (1-p_{15}))}$$

- Declare as spam if threshold > 0.9.

Details.

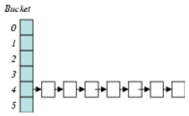
- Words you've never seen.
- Words that appear in ham corpus but not spam corpus, vice versa.
- Words that appear less than 5 times in spam and ham corpuses.
- Update data structures.

Algorithmic Complexity Attacks

Is the random hash map assumption important in practice?

- Yes, in obvious situations – aircraft control, nuclear reactors.
- Yes, sometimes in surprising situations.

Hashing-based denial-of-service attacks.



- If malicious adversary can choose what strings to insert into your hash table, you might be in big trouble.

Crosby-Wallach exploits of real systems.

- Bro server: send carefully chosen packets to DOS the server, using less bandwidth than a dial-up modem
- Perl 5.8.0: insert carefully chosen strings into associative array.
- Linux 2.4.20 kernel: save files with carefully chosen names.

Reference: `http://www.cs.rice.edu/~scrosby/hash/`

Algorithmic Complexity Attacks

How easy is it to break Java's `hashCode` with `String` keys?

- Almost trivial: String hash function is part of language spec.
- Java's string `hashCode`: hash of `"BB"` = hash of `"Aa"` = 2112.
- Can now create 2^N strings of length 2N that all hash to same value!

```
AaAaAaAa    BBAaAaAa
AaAaAaBB    BBAaAaBB
AaAaBBAa    BBAaBBAa
AaAaBBBB    BBAaBBBB
AaBBAaAa    BBBBAaAa
AaBBAaBB    BBBBAaBB
AaBBBBAa    BBBBBBAa
AaBBBBBB    BBBBBBBB
```

Possible to fix?

- Security by obscurity.
- Cryptographically secure hash functions.
- Universal hashing.