4.2 Hashing

Robert Sedgewick and Kevin Wayne • Copyright © 2006 • 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 with sequential search.
- Limitations on both time and space: hashing (the real world).

Choosing a Good Hash Function

Idealistic goal: scramble the keys uniformly.

- Efficiently computable.
- Each table position equally likely for each key.

thoroughly researched problem

Ex: Social Security numbers.

- Bad: first three digits. (573 = California, 574 = Alaska)
- Better: last three digits. (assigned in chronological order within a given geographic region)

Ex: date of birth.

- Bad: birth year.
- Better: birthday.

Ex: phone numbers.

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

Hash Codes and Hash Functions

Hash code. A 32-bit `int` (between `-2147483648` and `2147483647`).

Hash function. An `int` between `0` and `M-1`.

```
String s = "call";
int code = s.hashCode();
int hash = code % M;
```

7121 (M), 8191 (hash), 3045982 (code)

Bug. Don't use `(code % M)` as array index.
Subtle bug. Don't use `(Math.abs(code) % M)` as array index.

OK. Safe to use `((code & 0x7fffffff) % M)` as array index.

Implementing Hash Code in Java

API for `hashCode()`.

- Return an `int`.
- If `x.equals(y)` then `x` and `y` must have the same hash code.
- Repeated calls to `x.hashCode()` must return the same value.

inherited from Object

Default implementation. Memory address of `x`.
Customized implementations. `String`, `URL`, `Integer`, `Date`.

User-defined implementaitons. Tricky to get right, black art.

Designing a Good Hash Code

Java 1.5 string library.

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

ith character of s

char	Unicode
…	…
'a'	97
'b'	98
'c'	99
…	…

- Equivalent to $h = 31^{L-1} \cdot s_0 + \ldots + 31^2 \cdot s_{L-3} + 31 \cdot s_{L-2} + s_{L-1}$.
- Horner's method to hash string of length L: O(L).

Ex.

```
String s = "call";
int code = s.hashCode();
```

$3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$

Designing a Bad Hash Code

Java 1.1 string library.

- For long strings: only examines 8-9 evenly spaced characters.
- Saves time in performing arithmetic…

```
public int hashCode() {
   int hash = 0;
   int skip = Math.max(1, length() / 8);
   for (int i = 0; i < length(); i += skip)
      hash = (37 * hash) + s[i];
   return hash;
}
```

But great potential for bad collision patterns.

```
http://www.cs.princeton.edu/introcs/13loop/Hello.java
http://www.cs.princeton.edu/introcs/13loop/Hello.class
http://www.cs.princeton.edu/introcs/13loop/Hello.html
http://www.cs.princeton.edu/introcs/13loop/index.html
http://www.cs.princeton.edu/introcs/12type/index.html
```

Implementing Hash Code: US Phone Numbers

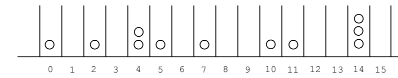
Phone numbers: (609) 867-5309.

area code, exchange, extension

```
public final class PhoneNumber {
   private final int area, exch, ext;

   public PhoneNumber(int area, int exch, int ext) {
      this.area = area;
      this.exch = exch;
      this.ext  = ext;
   }

   public boolean equals(Object y) { // as before }

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


Bins and Balls

Bins and balls. Throw balls uniformly at random into M bins.



Birthday problem. Expect two balls in the same bin after $\sqrt{\frac{1}{2}\pi M}$ tosses.

Coupon collector. Expect every bin has ≥ 1 ball after Θ(M ln M) tosses.

Load balancing. After tossing M balls, expect most loaded bin has Θ(log M / log log M) balls.

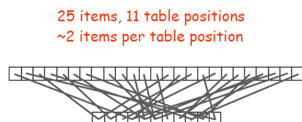
Collisions

Collision. Two distinct keys hashing to same index.

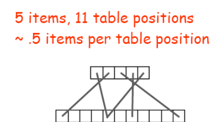
Conclusion. Birthday problem ⇒ can't avoid collisions unless you have a ridiculous amount of memory.

Challenge. Deal with collisions efficiently.

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



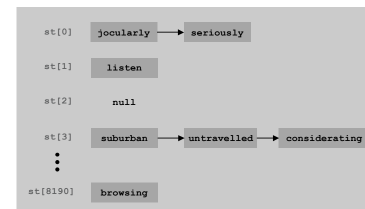
5 items, 11 table positions
~ .5 items per table position



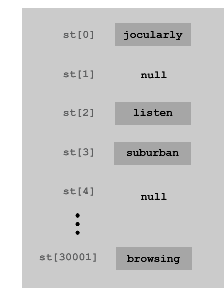
Collision Resolution: Two Approaches

Separate chaining. [H. P. Luhn, IBM 1953]
Put keys that collide in a list associated with index.

Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]
When a new key collides, find next empty slot, and put it there.



separate chaining (M = 8191, N = 15000)



linear probing (M = 30001, N = 15000)

Separate Chaining

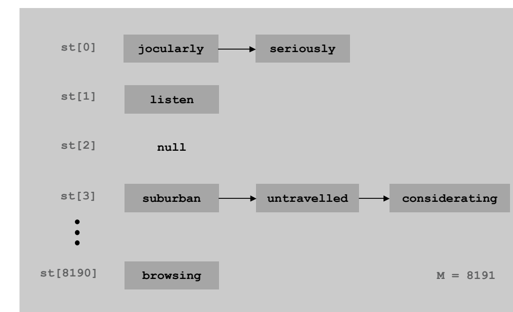
Robert Sedgewick and Kevin Wayne • Copyright © 2006 • http://www.Princeton.EDU/~cos226

Separate Chaining

typically M ≈ N/10

Separate chaining: array of M linked lists.

- Hash: map key to integer i between 0 and M-1.
- Insert: put at front of ith chain (if not already there).
- Search: only need to search i^{th} chain.



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

Separate Chaining: Java Implementation

```
public class ListHashST<Key, Value> {
   private int M = 8191;
   private Node[] st = new Node[M];

   private static class Node {
      Object key;
      Object val;
      Node   next;
      Node(Object key, Object val, Node next)  {
         this.key  = key;
         this.val  = val;
         this.next = next;
      }
   }

   private int hash(Key key) {
      return (key.hashCode() & 0x7fffffff) % M;
   }
```

no generic array creation in Java

between 0 and M-1

Separate Chaining: Java Implementation (cont)

```
public void put(Key key, Val val) {
   int i = hash(key);
   for (Node x = st[i]; x != null; x = x.next) {
      if (key.equals(x.key)) {
         x.val = val;
         return;
      }
   }
   st[i] = new Node(k, val, st[i]);
}

public Val get(Key key) {
   int i = hash(key);
   for (Node x = st[i]; x != null; x = x.next)
      if (key.equals(x.key))
         return (Val) x.val;
   return null;
}
```

check if key already present

insert at front of chain

Separate Chaining Performance

Separate chaining performance.

- Cost is proportional to length of chain.
- Average length = N / M.
- Worst case: all keys hash to same chain.

Theorem. Let α = N / M > 1 be average length of list. For any t > 1, probability that list length > t α 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: α = N / M $\approx$ 10 $\Rightarrow$ constant-time ops.

Symbol Table: Implementations Cost Summary

	Worst Case			Average Case		
Implementation	Get	Put	Remove	Get	Put	Remove
Sorted array	log N	N	N	log N	N/2	N/2
Unsorted list	N	N	N	N/2	N	N/2
Separate chaining	N	N	N	1*	1*	1*

* assumes hash function is random

Advantages. Fast insertion, fast search.
Disadvantage. Hash table has fixed size, assumes good hash function.

fix: use repeated doubling, and rehash all keys

Linear Probing

Robert Sedgewick and Kevin Wayne • Copyright © 2006 • http://www.Princeton.EDU/~cos226

Linear Probing

typically M $\approx$ 2N

Linear probing: array of size M.

- 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.

-	-	-	S	H	-	-	A	C	E	R	-	N
0	1	2	3	4	5	6	7	8	9	10	11	12

-	-	-	S	H	-	-	A	C	E	R	I	-
0	1	2	3	4	5	6	7	8	9	10	11	12

insert I
hash(I) = 11

-	-	-	S	H	-	-	A	C	E	R	I	N
0	1	2	3	4	5	6	7	8	9	10	11	12

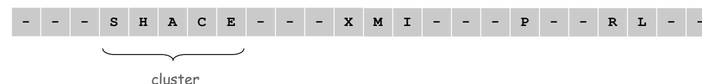
insert N
hash(N) = 8

Linear Probing: Java Implementation

```java
public class ArrayHashST<Key, Val> {
   private int M = 30001;
   private Key[] keys = (Key[]) new Object[M];   // ← no generic array creation in Java
   private Val[] vals = (Val[]) new Object[M];

   private int hash(Key key) { // as before }

   public void put(Key key, Val val) {
      int i;
      for (i = hash(key); keys[i] != null; i = (i+1) % M)
         if (keys[i].equals(key)) break;
      keys[i] = key;
      vals[i] = val;
   }

   public Val get(Key key) {
      for (int i = hash(key); keys[i] != null; i = (i+1) % M)
         if (keys[i].equals(key)) return vals[i];
      return null;
   }
}
```


Clustering

Cluster. A contiguous block of items.
Observation. New keys likely to hash into middle of big clusters.



Knuth's parking problem. Cars arrive at one-way street with M parking spaces. Each desires a random space i: if space i is taken, try i+1, i+2, … What is mean displacement of a car?

Empty. With M/2 cars, mean displacement is $\approx 3/2$.
Full. With M cars, mean displacement is $\approx \frac{1}{4}\sqrt{2\pi M}$.

Linear Probing Performance

Linear probing performance.

- Insert and search cost depend on length of cluster.
- Average length of cluster = α = N / M. ← but keys more likely to hash to big clusters
- Worst case: all keys hash to same cluster.

Theorem. [Knuth 1962] Let α = N / M < 1 be the load factor.

$$\text{insert / search miss} \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

$$\text{search hit} \approx \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right)$$

← assumes hash function is random

Parameters.

- M too large ⇒ too many empty array entries.
- M too small ⇒ clusters coalesce.
- Typical choice: M ≈ 2N ⇒ constant-time ops.

Symbol Table: Implementations Cost Summary

	Worst Case			Average Case		
Implementation	Get	Put	Remove	Get	Put	Remove
Sorted array	log N	N	N	log N	N / 2	N / 2
Unsorted list	N	N	N	N / 2	N	N / 2
Separate chaining	N	N	N	1*	1*	1*
Linear probing	N	N	N	1*	1*	1*

* assumes hash function is random

Advantages. Fast insertion, fast search.
Disadvantage. Hash table has fixed size, assumes good hash function.

← fix: use repeated doubling, and rehash all keys

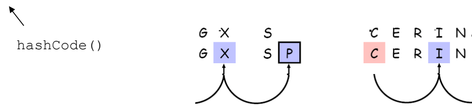
Double Hashing

Idea 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 k.

Ex: k = 1 + (v mod 97).

hashCode()



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

Best practices. Make k and M relatively prime.

Double Hashing Performance

Theorem. [Guibas-Szemerédi] Let $\alpha = N / M < 1$ be average length of list.

insert / search miss $\approx \frac{1}{1 - \alpha}$

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

assumes hash function is random

Parameters. Typical choice: $M \approx 2N \Rightarrow$ constant-time ops.

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	get	1.5	2.0	3.0	5.5
	put	2.5	5.0	8.5	55.5
double hashing	get	1.4	1.6	1.8	2.6
	put	1.5	2.0	3.0	5.5

number of probes

Odds and Ends

Robert Sedgewick and Kevin Wayne • Copyright © 2006 • http://www.Princeton.EDU/~cos226

Hashing: Java Library

Java has built-in libraries for symbol tables.

- `java.util.HashMap` = linear probing hash table implementation.

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

Duplicate policy.

- Java `HashMap` allows `null` values.
- Our implementation forbids `null` values.

Symbol Table: Using HashMap

Symbol table. Implement our API using `java.util.HashMap`.

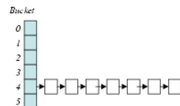
```
import java.util.HashMap;
import java.util.Iterator;

public class ST<Key, Val> implements Iterable<Key> {
   private HashMap<Key, Val> st = new HashMap<Key, Val>();

   public void put(Key key, Val val) {
      if (val == null) st.remove(key);
      else             st.put(key, val);
   }
   public Val get(Key key)           { return st.get(key);            }
   public Val remove(Key key)        { return st.remove(key);         }
   public boolean contains(Key key)  { return st.containsKey(key);    }
   public int size()                 { return st.size();              }
   public Iterator<Key> iterator()   { return st.keySet().iterator(); }
}
```


Algorithmic Complexity Attacks

Is the random hash map assumption important in practice?

- Obvious situations: aircraft control, nuclear reactor, pacemaker.
- Surprising situations: denial-of-service attacks.

malicious adversary learns your ad hoc hash function (e.g., by reading Java API) and causes a big pile-up in single address that grinds performance to a halt



Real-world exploits. [Crosby-Wallach 2003]

- 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 Attack: Java Library

Goal. Find strings with the same hash code.
Solution. The base-31 hash code is part of Java's string API.

Key	hashCode()
Aa	2112
BB	2112

Key	hashCode()
AaAaAaAa	-540425984
AaAaAaBB	-540425984
AaAaBBAa	-540425984
AaAaBBBB	-540425984
AaBBAaAa	-540425984
AaBBAaBB	-540425984
AaBBBBAa	-540425984
AaBBBBBB	-540425984
BBAaAaAa	-540425984
BBAaAaBB	-540425984
BBAaBBAa	-540425984
BBAaBBBB	-540425984
BBBBAaAa	-540425984
BBBBAaBB	-540425984
BBBBBBAa	-540425984
BBBBBBBB	-540425984

2^N strings of length 2N that hash to same value!

One-Way Hash Functions

One-way hash function. Hard to find a key that will hash to a desired value, or to find two keys that hash to same value.

Ex. MD4, MD5, SHA-0, SHA-1, SHA-2, WHIRLPOOL, RIPEMD-160.

```
String password = args[0];
MessageDigest sha1 = MessageDigest.getInstance("SHA1");
byte[] bytes = sha1.digest(password);

// prints bytes as hex string
```

Applications. Digital fingerprint, message digest, storing passwords.