4.2 Hashing

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.
- 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 Function: String Keys

Java string library hash functions.

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

ith character of s

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

7121 8191 3045982

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

Q. Can we reliably use `(h % M)` as index for table of size M?
A. No. Instead, use `(h & 0x7fffffff) % M`.

hashCode

Hash code. For any references `x` and `y`:

- Repeated calls to `x.hashCode()` must return the same value provided no information used in `equals` comparison has changed.
- If `x.equals(y)` then `x` and `y` must have the same hash code.

"consistent with equals"

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

Implementing HashCode: 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;
   }
}
```


Collisions

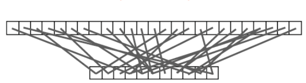
Collision = two keys hashing to same value.

- 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 $\text{sqrt}(\frac{1}{2} \pi M)$ insertions.

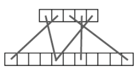
Conclusion: can't avoid collisions unless you have a ridiculous amount of memory.

Challenge: efficiently cope with collisions.

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

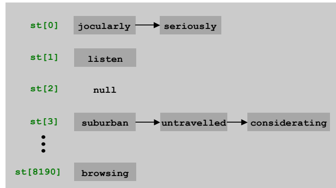


5 items, 11 table positions
~ .5 items per table position



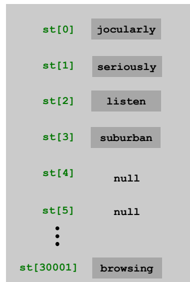
Collision Resolution: Two Approaches.

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



M = 8191, N = 15000

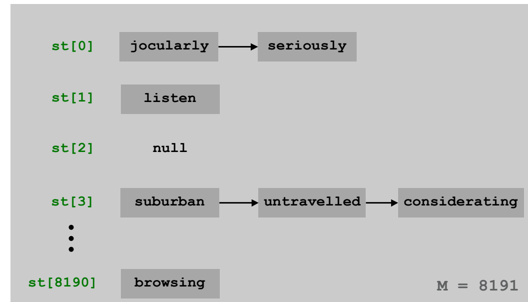
Open addressing.
- M much larger than N.
- Plenty of empty table slots.
- When a new key collides, find next empty slot and put it there.
- Complex collision patterns.



M = 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 (if not already there).
- Search: only need to search i^{th} chain.
- Running time: proportional to length of chain.



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

Symbol Table: Separate Chaining

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

   private static class Node {
      Object key;
      Object val;     no generic array creation in Java
      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;
   }                            hex
                                          between 0 and M-1
```


Symbol Table: Separate Chaining Implementation (cont)

```
public void put(Key key, Value val) {
   int i = hash(key);
   for (Node x = st[i]; x != null; x = x.next) {
      if (key.equals(x.key)) {
         x.val = val;
         return;                check if key already present
      }
   }         insert at front of chain
   st[i] = new Node(k, val, st[i]);
}

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

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 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 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	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.

fix: use repeated doubling, and rehash all keys

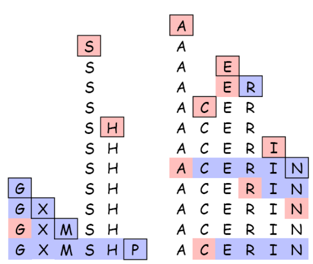
Linear Probing

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.



A S E A R C H I N G X M P

Symbol Table: Linear Probing Implementation

```
public class ArrayHashST<Key, Val> {
   private int M = 30001;
   private Key[] keys = (Key[]) new Object[M];
   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) {
      int i;
      for (i = hash(key); keys[i] != null; i = (i+1) % M)
         if (keys[i].equals(key)) break;
      return vals[i];
   }
}
```

no generic array creation in Java

Linear Probing Performance

Linear probing performance.

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

Theorem. [Knuth 1962] Let α = N / M < 1 be average length of list.

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

← depends on hash map being random map

Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ clusters coalesce.
- Typical choice: M $\approx$ 2N $\Rightarrow$ 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	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.

fix: use repeated doubling, and rehash all keys

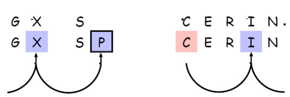
Double Hashing

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 k.

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

hashCode



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

Best practices. Make k and M relatively prime.

Double Hashing Performance

Linear probing performance.

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

Theorem. [Guibas-Szemeredi] Let α = N / M < 1 be average length of list.

$$\text{insert: } \frac{1}{1-\alpha} \qquad \text{search: } \frac{1}{\alpha}\ln(1+\alpha)$$

← depends on hash map being random map

Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ clusters coalesce.
- Typical choice: M $\approx$ 2N $\Rightarrow$ 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

Hash Table: Java Library

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<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 implementations forbid `null` values.

Symbol Table: Using HashMap

Symbol table. Implement our interface using `HashMap`.

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

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

   public void put(Key key, Value val) {
      if (val == null) st.remove(key);
      else             st.put(key, val);
   }
   public Value get(Key key)         { return st.get(key);              }
   public Value 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();   }
}
```


Designing a Good Hash Function

Java 1.1 string library hash function.

- For long strings: only examines 8 evenly spaced characters.
- Saves time in performing arithmetic.
- Great potential for bad collision patterns.

```
public int hashCode() {
   int hash = 0;
   if (length() < 16) {
      for (int i = 0; i < length(); i++)
         hash = (37 * hash) + s[i];
   }
   else {
      int skip = length() / 8;
      for (int i = 0; i < length(); i += skip)
         hash = (37 * hash) + s[i];
   }
   return hash;
}
```

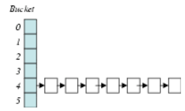
String.java

Algorithmic Complexity Attacks

Is the random hash map assumption important in practice?

- Obvious situations: aircraft control, nuclear reactors.
- 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 Attacks

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

A. Almost trivial: string `hashCode` is part of Java 1.5 API.

- Ex: `hashCode` of `"BB"` equals `hashCode` of `"Aa"`.
- 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. [not recommended]
- Cryptographically secure hash functions.
- Universal hashing.