Hashing

- hash functions
- collision resolution
- applications

References:

Algorithms in Java, Chapter 14

http://www.cs.princeton.edu/introalgsds/42hash

Summary of symbol-table implementations

implementation	guarantee			average case			ordered iteration?
	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.39 lg N	1.39 lg N	1.39 lg N	yes
red-black tree	3 lg N	3 lg N	3 lg N	lg N	lg N	lg N	yes

Can we do better?

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

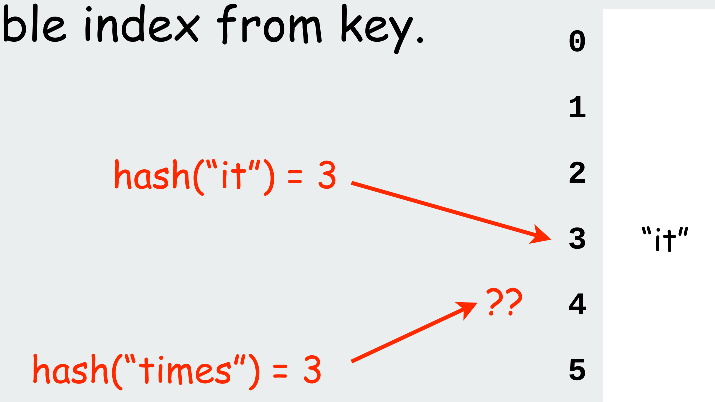
> We follow two rules in the matter of optimization:
>
> Rule 1: Don't do it.
>
> Rule 2 (for experts only). Don't do it yet - that is, not until you have a perfectly clear and unoptimized solution.
>
> - M. A. Jackson

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.



Issues.

1. Computing the hash function
2. Collision resolution: Algorithm and data structure to handle two keys that hash to the same index.
3. Equality test: Method for checking whether two keys are equal.

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).

- hash functions
- collision resolution
- applications

Computing the hash function

Idealistic goal: scramble the keys uniformly.

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

thoroughly researched problem, still problematic in practical applications

Practical challenge: need different approach for each type of key

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: birth year.
- Better: birthday.

Ex: phone numbers.

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

Hash Codes and Hash Functions

Java convention: all classes implement **hashCode()**

hashcode() returns a 32-bit **int** (between **-2147483648** and **2147483647**)

Hash function. An **int** between **0** and **M-1** (for use as an array index)

First try:

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

3045982 (value of s.hashCode()); 7121 (hash); 8191 (M)

Bug. Don't use **(code % M)** as array index

1-in-a billion bug. Don't use **(Math.abs(code) % M)** as array index.

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

hex literal 31-bit mask

Java's `hashCode()` convention

Theoretical advantages

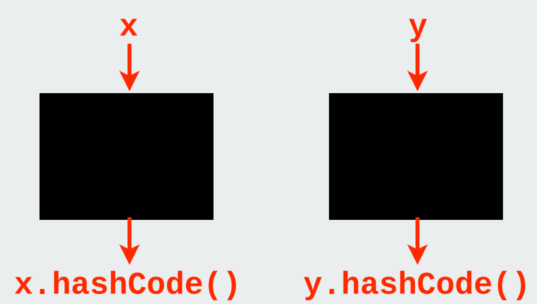
- Ensures hashing can be used for every type of object
- Allows expert implementations suited to each type

Requirements:

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

Practical realities

- True randomness is hard to achieve
- Cost is an important consideration



Available implementations

- default (inherited from Object): Memory address of x (! ! !)
- customized Java implementations: `String`, `URL`, `Integer`, `Date`.
- User-defined types: users are on their own

that's you!

A typical type

Assumption when using hashing in Java:

Key type has reasonable implementation of **hashCode()** and **equals()**

Ex. Phone numbers: (609) 867-5309.

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;  }
}
```

← sufficiently random?

Fundamental problem:

Need a theorem for each data type to ensure reliability.

A decent hash code design

Java 1.5 string library [see also Program 14.2 in Algs in Java].

```
public int hashCode()
{
   int hash = 0;
   for (int i = 0; i < length(); i++)
      hash = s[i] + (31 * hash);
   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: L multiplies/adds

Ex.

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

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

$= 108 + 31 \cdot (108 + 31 \cdot (99 + 31 \cdot (97)))$

Provably random? Well, no.

A poor hash code design

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
```

Basic rule: need to use the whole key.

Digression: using a hash function for data mining

Use content to characterize documents.

Applications

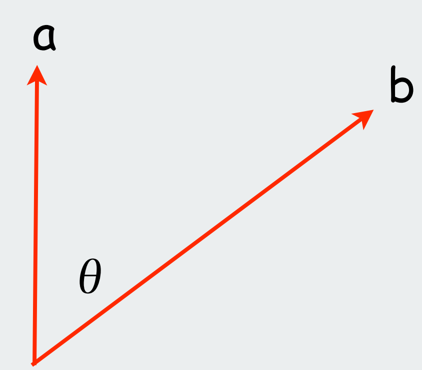
- Search documents on the web for documents similar to a given one.
- Determine whether a new document belongs in one set or another

Approach

- Fix order k and dimension d
- Compute `hashCode() % d` for all k-grams in the document
- Result: d-dimensional vector profile of each document
- To compare documents: Consider angle θ separating vectors

  $\cos \theta$ close to 0: not similar

  $\cos \theta$ close to 1: similar



$$\cos \theta = a \cdot b \, / \, |a| \, |b|$$

Digression: using a hash function for data mining

k = 10
d = 65536

```
% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of
foolishness
...
```

```
% more genome.txt
CTTTCGGTTTGGAACC
GAAGCCGCGCGTCT
TGTCTGCTGCAGC
ATCGTTC
...
```

	tale.txt		genome.txt	
i	10-grams with hashcode() i	freq	10-grams with hashcode() i	freq
0		0		0
1		0		0
2		0		0
435	best of ti foolishnes	2	TTTCGGTTTG TGTCTGCTGC	2
8999	it was the	8		0
...				
12122		0	CTTTCGGTTT	3
...				
34543	t was the b	5	ATGCGGTCGA	4
...				
65535				
65536				

profiles

$\cos\ \theta$ small: not similar

Digression: using a hash function to profile a document for data mining

```
public class Document
{
   private String name;
   private double[] profile;
   public Document(String name, int k, int d)
   {
      this.name = name;
      String doc = (new In(name)).readAll();
      int N = doc.length();
      profile = new double[d];
      for (int i = 0; i < N-k; i++)
      {
         int h = doc.substring(i, i+k).hashCode();
         profile[Math.abs(h % d)] += 1;
      }
   }
   public double simTo(Document other)
   {
      // compute dot product and divide by magnitudes
   }
}
```

Digression: using a hash function to compare documents

```
public class CompareAll
{
   public static void main(String args[])
   {
      int k = Integer.parseInt(args[0]);
      int d = Integer.parseInt(args[1]);
      int N = StdIn.readInt();
      Document[] a = new Document[N];
      for (int i = 0; i < N; i++)
         a[i] = new Document(StdIn.readString(), k, d);
      System.out.print("      ");
      for (int j = 0; j < N; j++)
         System.out.printf("    %.4s", a[j].name());
      System.out.println();
      for (int i = 0; i < N; i++)
      {
         System.out.printf("%.4s  ", a[i].name());
         for (int j = 0; j < N; j++)
            System.out.printf("%8.2f", a[i].simTo(a[j]));
         System.out.println();
      }
   }
}
```

Digression: using a hash function to compare documents

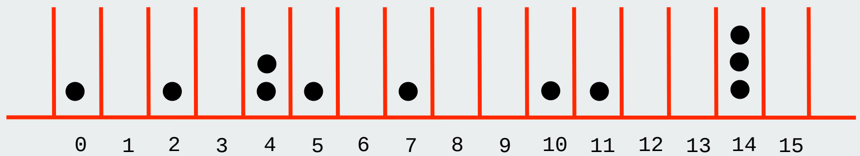
Cons	US Constitution
TomS	"Tom Sawyer"
Huck	"Huckleberry Finn"
Prej	"Pride and Prejudice"
Pict	a photograph
DJIA	financial data
Amaz	Amazon.com website .html source
ACTG	genome

```
% java CompareAll 5 1000 < docs.txt
          Cons    TomS    Huck    Prej    Pict    DJIA    Amaz    ACTG
Cons      1.00    0.89    0.87    0.88    0.35    0.70    0.63    0.58
TomS      0.89    1.00    0.98    0.96    0.34    0.75    0.66    0.62
Huck      0.87    0.98    1.00    0.94    0.32    0.74    0.65    0.61
Prej      0.88    0.96    0.94    1.00    0.34    0.76    0.67    0.63
Pict      0.35    0.34    0.32    0.34    1.00    0.29    0.48    0.24
DJIA      0.70    0.75    0.74    0.76    0.29    1.00    0.62    0.58
Amaz      0.63    0.66    0.65    0.67    0.48    0.62    1.00    0.45
ACTG      0.58    0.62    0.61    0.63    0.24    0.58    0.45    1.00
```

- hash functions
- **collision resolution**
- applications

Helpful results from probability theory

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



Birthday problem.

Expect two balls in the same bin after $\sqrt{\pi M / 2}$ tosses.

Coupon collector.

Expect every bin has ≥ 1 ball after $\Theta(M \ln M)$ tosses.

Load balancing.

After M tosses, expect most loaded bin has $\Theta(\log M / \log \log M)$ balls.

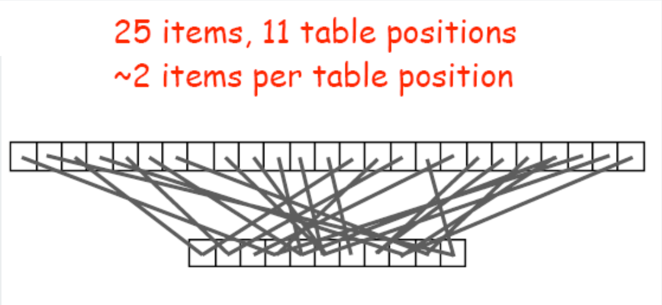
Collisions

Collision. Two distinct keys hashing to same index.

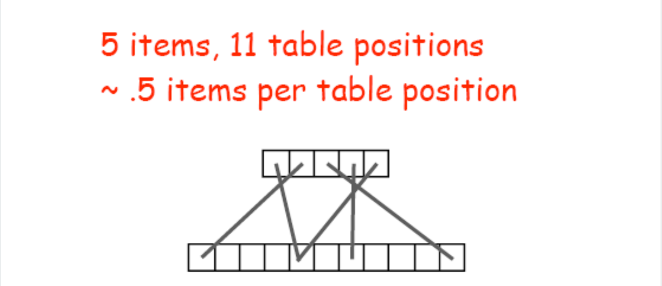
Conclusion. Birthday problem $\Rightarrow$ can't avoid collisions unless you have a ridiculous amount of memory.

Challenge. Deal with collisions efficiently.

Approach 1:
accept multiple collisions



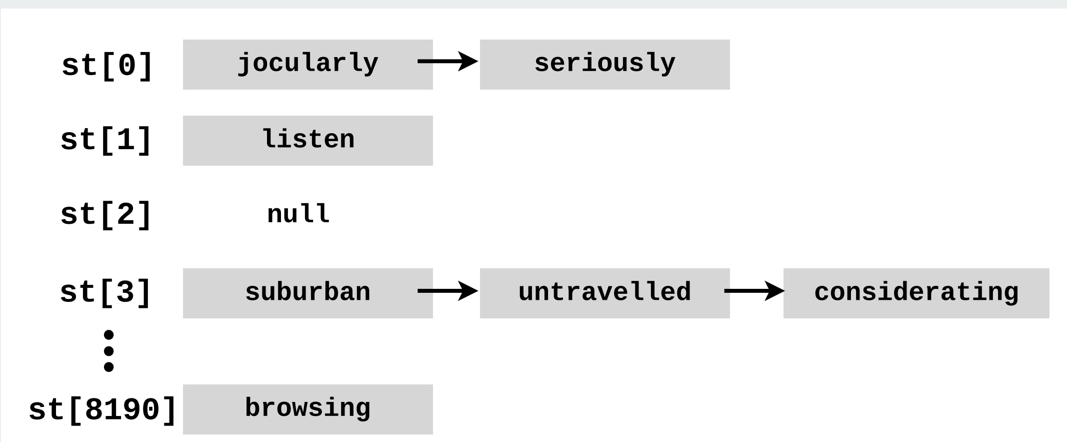
Approach 2:
minimize collisions



Collision resolution: two approaches

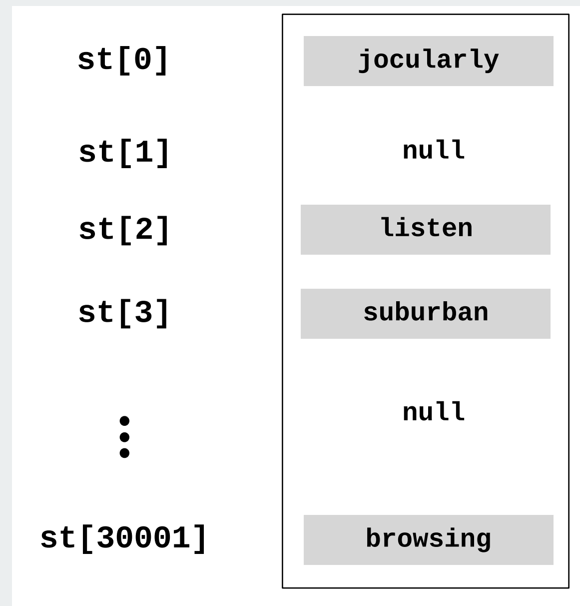
1. **Separate chaining.** [H. P. Luhn, IBM 1953]
Put keys that collide in a list associated with index.

2. **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)

easy extension of linked list ST implementation



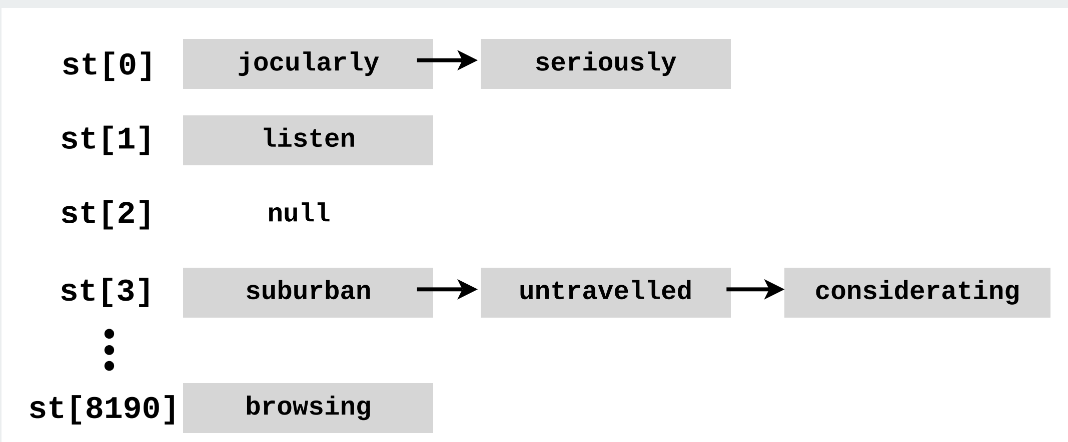
linear probing (M = 30001, N = 15000)

easy extension of array ST implementation

Collision resolution approach 1: separate chaining

Use an array of M < N linked lists. ← good choice: M ≈ N/10

- 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.



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

Separate chaining ST implementation (skeleton)

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

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

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

  public void put(Key key, Value val)
  // see next slide

  public Val get(Key key)
  // see next slide
}
```

could use doubling → (M = 8191)

no generics in arrays in Java → (Object key; Object val;)

compare with linked lists

Separate chaining ST implementation (put and get)

```
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; }
   st[i] = new Node(key, value, first);
}
```

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

Identical to linked-list code, except hash to pick a list.

Separate chaining performance.

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

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: $\alpha = N / M \approx 10$ $\Rightarrow$ constant-time ops.

Collision resolution approach 2: open addressing

Use an array of size M >> N. ← good choice: M ≈ 2N

- Hash: map key to integer i between 0 and M-1.

Linear probing:

- 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 ST implementation

compare with elementary unordered array implementation

```
public class ArrayHashST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[maxN];
   private Key[]   keys = (Key[]) new Object[maxN];

   privat int hash(Key key) // as before

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

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

standard ugly casts

standard array doubling code omitted (double when half full)

Clustering

Cluster. A contiguous block of items.

Observation. New keys likely to hash into middle of big clusters.

-	-	-	S	H	A	C	E	-	-	-	X	M	I	-	-	-	P	-	-	R	L	-	-

cluster

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 about 3/2.

Full. Mean displacement for the last car is about $\sqrt{\pi M / 2}$

Analysis of linear probing

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 $\alpha = N / M < 1$ be the load factor.

Average probes for insert/search miss

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right) = (1 + \alpha + 2\alpha^2 + 3\alpha^3 + 4\alpha^4 + \ldots)/2$$

Average probes for search hit

$$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)}\right) = 1 + (\alpha + \alpha^2 + \alpha^3 + \alpha^4 + \ldots)/2$$

Parameters.

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

Hashing: variations on the theme

Many improved versions have been studied:

Ex: Two-probe hashing

- hash to two positions, put key in shorter of the two lists
- reduces average length of the longest list to log log N

Ex: Double hashing

- use linear probing, but skip a variable amount, not just 1 each time
- effectively eliminates clustering
- can allow table to become nearly full

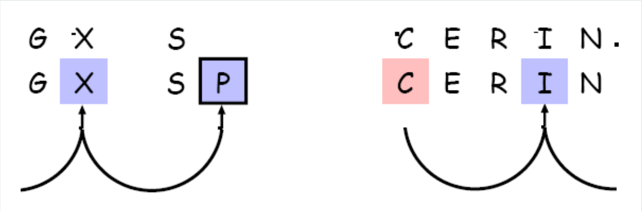
Double hashing

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.

Average probes for insert/search miss

$$\frac{1}{(1-\alpha)} = 1 + \alpha + \alpha^2 + \alpha^3 + \alpha^4 + \ldots$$

Average probes for search hit

$$\frac{1}{\alpha} \ln \frac{1}{(1-\alpha)} = 1 + \alpha/2 + \alpha^2/3 + \alpha^3/4 + \alpha^4/5 + \ldots$$

Parameters. Typical choice: $\alpha \approx 1.2 \Rightarrow$ 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

Summary of symbol-table implementations

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search	insert	delete		
unordered array	N	N	N	N/2	N/2	N/2	no	`equals()`
ordered array	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
unordered list	N	N	N	N/2	N	N/2	no	`equals()`
ordered list	N	N	N	N/2	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
randomized BST	7 lg N	7 lg N	7 lg N	1.38 lg N	1.38 lg N	1.38 lg N	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	lg N	lg N	lg N	yes	`compareTo()`
hashing	1*	1*	1*	1*	1*	1*	no	`equals()` `hashCode()`

* assumes random hash code

Hashing versus balanced trees

Hashing

- simpler to code
- no effective alternative for unordered keys
- faster for simple keys (a few arithmetic ops versus lg N compares)
- (Java) better system support for strings [cached hashcode]
- does your hash function produce random values for your key type??

Balanced trees

- stronger performance guarantee
- can support many more operations for ordered keys
- easier to implement `compareTo()` correctly than `equals()` and `hashCode()`

Java system includes both

- red-black trees: `java.util.TreeMap`, `java.util.TreeSet`
- hashing: `java.util.HashMap`, `java.util.IdentityHashMap`

Typical "full" ST API

`public class *ST<Key extends Comparable<Key>, Value>`

	`*ST()`	create a symbol table
`void`	`put(Key key, Value val)`	put key-value pair into the table
`Value`	`get(Key key)`	return value paired with **key** (**null** if **key** is not in table)
`boolean`	`contains(Key key)`	is there a value paired with **key**?
`Key`	`min()`	smallest key
`Key`	`max()`	largest key
`Key`	`next(Key key)`	next largest key (**null** if **key** is max)
`Key`	`prev(Key key)`	next smallest key (**null** if **key** is min)
`void`	`remove(Key key)`	remove key-value pair from table
`Iterator<Key>`	`iterator()`	iterator through keys in table

Hashing is not suitable for implementing such an API (no order)

BSTs are easy to extend to support such an API (basic tree ops)

Ex: Can use LLRB trees implement priority queues for distinct keys

- hash functions
- collision resolution
- **applications**

Set ADT

Set. Collection of distinct keys.

public class *SET<Key extends Comparable<Key>, Value>		
	`SET()`	create a set
`void`	`add(Key key)`	put key into the set
`boolean`	`contains(Key key)`	is there a value paired with **key**?
`void`	`remove(Key key)`	remove key from the set
`Iterator<Key>`	`iterator()`	iterator through all keys in the set

Normal mathematical assumption: collection is unordered
Typical (eventual) client expectation: ordered iteration

Q. How to implement?
A0. Hashing (our ST code [value removed] or **java.util.HashSet**)
A1. Red-black BST (our ST code [value removed] or **java.util.TreeSet**)

unordered iterator
O(1) search

ordered iterator
O(log N) search

SET client example 1: dedup filter

Remove duplicates from strings in standard input

- Read a key.
- If key is not in set, insert and print it.

No iterator needed.
Output is in same order
as input with
dups removed.

```
public class DeDup
{
   public static void main(String[] args)
   {
      SET<String> set = new SET<String>();
      while (!StdIn.isEmpty())
      {
         String key = StdIn.readString();
         if (!set.contains(key))
         {
            set.add(key);
            StdOut.println(key);
         }
      }
   }
}
```

```
% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of
foolishness
...

% java Dedup < tale.txt
it
was
the
best
of
times
worst
age
wisdom
foolishness
...
```

Simplified version of **FrequencyCount** (no iterator needed)

SET client example 2A: lookup filter

Print words from standard input that are found in a list

- Read in a list of words from one file.
- Print out all words from standard input that are in the list.

```
public class LookupFilter
{
   public static void main(String[] args)
   {
      SET<String> set = new SET<String>();

      In in = new In(args[0]);
      while (!in.isEmpty())
         set.add(in.readString());

      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (set.contains(word))
            StdOut.println(word);
      }
   }
}
```

create SET

process list

print words that are not in list

SET client example 2B: exception filter

Print words from standard input that are not found in a list

- Read in a list of words from one file.
- Print out all words from standard input that are not in the list.

```
public class LookupFilter
{
   public static void main(String[] args)
   {
      SET<String> set = new SET<String>();

      In in = new In(args[0]);
      while (!in.isEmpty())
         set.add(in.readString());

      while (!StdIn.isEmpty())
      {
         String word = StdIn.readString();
         if (!set.contains(word))
            StdOut.println(word);
      }
   }
}
```

create SET

process list

print words that are not in list

SET filter applications

application	purpose	key	type	in list	not in list
dedup	eliminate duplicates		dedup	duplicates	unique keys
spell checker	find misspelled words	word	exception	dictionary	misspelled words
browser	mark visited pages	URL	lookup	visited pages	
chess	detect draw	board	lookup	positions	
spam filter	eliminate spam	IP addr	exception	spam	good mail
trusty filter	allow trusted mail	URL	lookup	good mail	
credit cards	check for stolen cards	number	exception	stolen cards	good cards

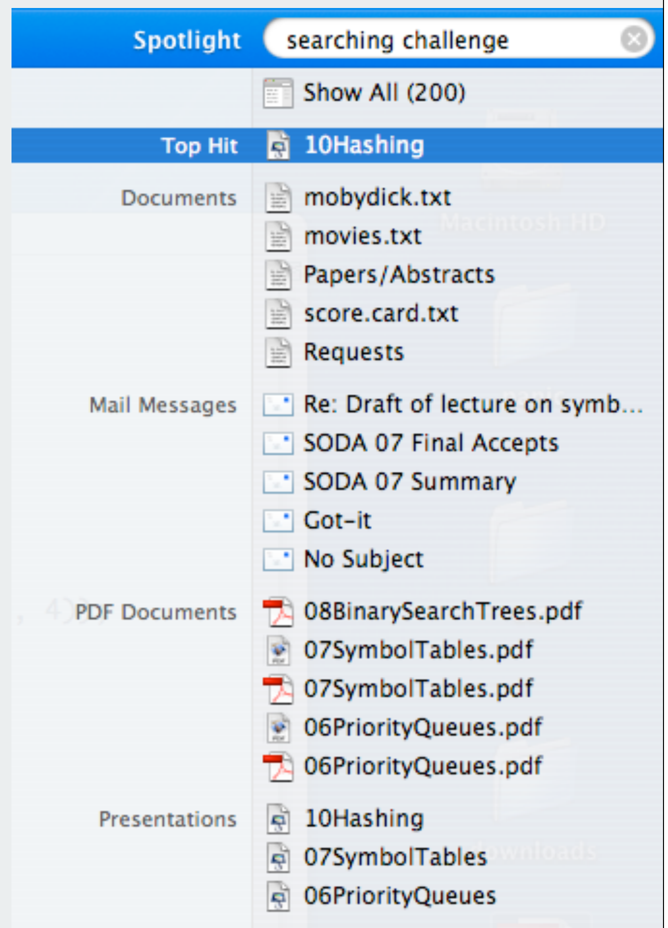
Searching challenge:

Problem: Index for a PC or the web

Assumptions: 1 billion++ words to index

Which searching method to use?

1) hashing implementation of SET
2) hashing implementation of ST
3) red-black-tree implementation of ST
4) red-black-tree implementation of SET
5) doesn't matter much



Index for search in a PC

```
ST<String, SET<File>> st = new ST<String, SET<File>>();
for (File f: filesystem)
{
   In in = new In(f);
   String[] words = in.readAll().split("\\s+");
   for (int i = 0; i < words.length; i++)
   {
       String s = words[i];
       if (!st.contains(s))
          st.put(s, new SET<File>());
       SET<File> files = st.get(s);
       files.add(f);
   }
}
```

← build index

```
SET<File> files = st.get(s);
for (File f: files) ...
```

← process lookup request

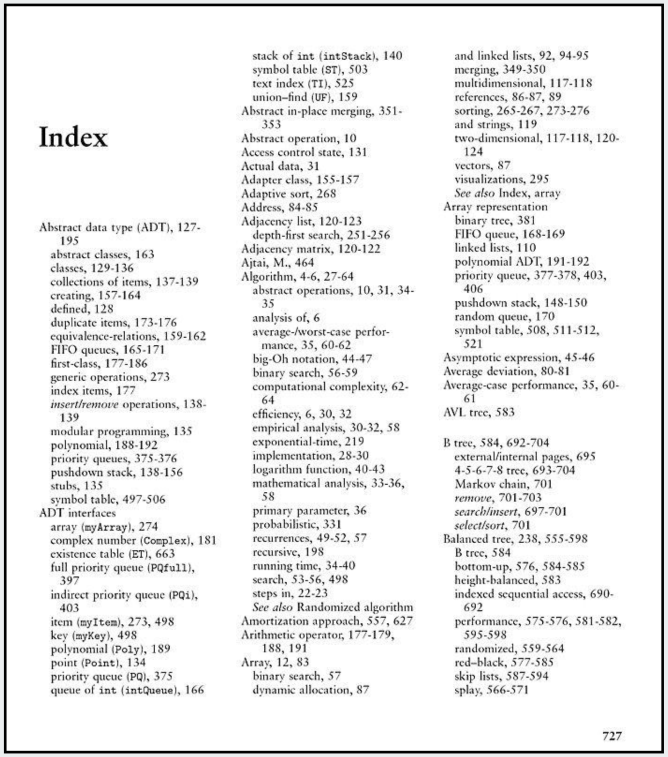
Searching challenge:

Problem: Index for a book

Assumptions: book has 100,000+ words

Which searching method to use?

1) hashing implementation of SET
2) hashing implementation of ST
3) red-black-tree implementation of ST
4) red-black-tree implementation of SET
5) doesn't matter much



Index

727

Index for a book

```
public class Index
{
   public static void main(String[] args)
   {
      String[] words = StdIn.readAll().split("\\s+");
      ST<String, SET<Integer>> st;
      st = new ST<String, SET<Integer>>();

      for (int i = 0; i < words.length; i++)
      {
         String s = words[i];
         if (!st.contains(s))
            st.put(s, new SET<Integer>());
         SET<Integer> pages = st.get(s);
         pages.add(page(i));
      }

      for (String s : st)
         StdOut.println(s + ": " + st.get(s));

   }
}
```

read book and create ST

process all words

print index!

Requires ordered iterators (not hashing)

Hashing in the wild: Java implementations

Java has built-in libraries for hash tables.

- **java.util.HashMap** = separate chaining implementation.
- **java.util.IdentityHashMap** = linear probing 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");
      StdOut.println(st.get("www.cs.princeton.edu"));
   }
}
```

Null value policy.

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

Using **HashMap**

Implementation of our API with **java.util.HashMap**.

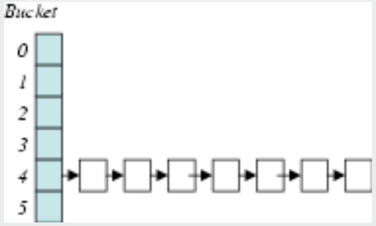
```
import java.util.HashMap;
import java.util.Iterator;

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.contains(key);       }
   public int size() contains(Key key) { return st.size();              }
   public Iterator<Key> iterator()      { return st.keySet().iterator(); }
}
```

Hashing in the wild: 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 on the 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!

Does your hash function produce random values for your key type??

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.

insecure

```
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.

Too expensive for use in ST implementations (use balanced trees)