Hashing

References:

Algorithms in Java, Chapter 14

`http://www.cs.princeton.edu/algs4/44hash`

- hash functions
- separate chaining
- linear probing
- applications

Algorithms in Java, 4th Edition · Robert Sedgewick and Kevin Wayne · Copyright © 2008 · January 30, 2009 11:16:15 AM

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

ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`

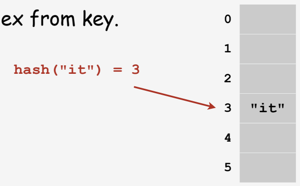
Q. Can we do better?

A. Yes, but with different access to the data.

Hashing: basic plan

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing array index from key.



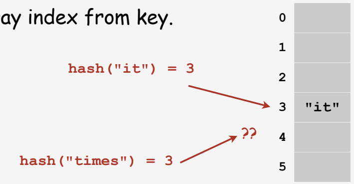
Issues.

- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.

Hashing: basic plan

Save items in a key-indexed table (index is a function of the key).

Hash function. Method for computing array index from key.



Issues.

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

Classic space-time tradeoff.

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

- **hash functions**
- separate chaining
- linear probing
- applications

Equality test

Needed because hash methods do not use `compareTo()`.

All Java classes have a method `equals()`, inherited from `Object`.

Java requirements. For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is `true`.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is `false`.

do `x` and `y` refer to the same object?

Default implementation (inherited from `Object`). `(x == y)`

Customized implementations. `Integer`, `Double`, `String`, `URI`, `Date`, …

User-defined implementations. Some care needed.

Implementing equals for user-defined types

Seems easy

```
public       class Record
{
   private final String name;
   private final int id;
   private final double value;
   ...

   public boolean equals(Record y)
   {






      Record that =          y;
      return (this.id == that.id) &&
             (this.value == that.value) &&
             (this.equals(that.name));
   }
}
```

check that all significant fields are the same

Implementing equals for user-defined types

Seems easy, but requires some care.

```
public final class Record
{
   private final String name;
   private final int id;
   private final double value;
   ...

   public boolean equals(Object y)
   {
      if (y == this) return true;

      if (y == null) return false;

      if (y.getClass() != this.getClass())
         return false;

      Record that = (Record) y;
      return (this.id == that.id) &&
             (this.value == that.value) &&
             (this.equals(that.name));
   }
}
```

no safe way to use `equals()` with inheritance

must be `Object`. Why? Experts still debate.

optimize for true object equality

check for `null`

objects must be in the same class

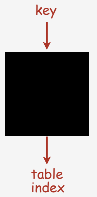
check that all significant fields are the same

Computing the hash function

Idealistic goal. Scramble the keys uniformly to produce a table index.

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

thoroughly researched problem, still problematic in practical applications



Ex 1. Phone numbers.

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

Ex 2. Social Security numbers.

573 = California, 574 = Alaska (assigned in chronological order within geographic region)

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

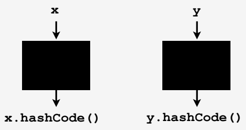
Practical challenge. Need different approach for each key type.

Java's hash code conventions

All Java classes have a method `hashCode()`, which returns an `int`.

Requirement. If `x.equals(y)`, then `(x.hashCode() == y.hashCode())`.

Highly desirable. If `!x.equals(y)`, then `(x.hashCode() != y.hashCode())`.



Default implementation (inherited from `Object`). Memory address of `x`.

Customized implementations. `Integer`, `Double`, `String`, `URI`, `Date`, …

User-defined types. Users are on their own.

Implementing hash code: integers and doubles

```
public final class Integer
{
   private final int value;
   ...

   public int hashCode()
   {  return value;  }
}
```

```
public final class Double
{
   private final double value;
   ...

   public int hashCode()
   {
      long bits = doubleToLongBits(value);
      return (int)(bits ^ (bits >>> 32));
   }
}
```

convert to IEEE 64-bit representation;
xor most significant 32-bits
with least significant 32-bits

Implementing hash code: strings

```
public final class String
{
   private final char[] s;
   ...

   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
…	…

- Horner's method to hash string of length L: L multiplies/adds.
- Equivalent to $h = 31^{L-1} \cdot s^0 + \ldots + 31^2 \cdot s^{L-3} + 31^1 \cdot s^{L-2} + 31^0 \cdot s^{L-1}$.

Ex.

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

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

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

A poor hash code

Ex. Strings (in Java 1.1).

- For long strings: only examine 8-9 evenly spaced characters.
- Benefit: 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 = s[i] + (37 * hash);
   return hash;
}
```

- Downside: 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: user-defined types

```
public final class Record
{
   private String name;
   private int id;
   private double value;

   public Record(String name, int id, double value)
   {  /* as before */  }

   ...

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

   public int hashCode()
   {
      int hash = 17;
      hash = 31*hash + name.hashCode();
      hash = 31*hash + id;
      hash = 31*hash + Double.valueOf(value).hashCode();
      return hash;
   }
}
```

nonzero constant

typically a small prime

Hash code design

"Standard" recipe for user-defined types.

- Combine each significant field using the 31x + y rule.
- If field is a primitive type, use built-in hash code.
- If field is an array, apply to each element.
- If field is an object, apply rule recursively.

In practice. Recipe works reasonably well; used in Java libraries.

In theory. Need a theorem for each type to ensure reliability.

Basic rule. Need to use the whole key to compute hash code; consult an expert for state-of-the-art hash codes.

Hash functions

Hash code. An `int` between -2^{31} and $2^{31}-1$.

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

typically a prime or power of 2

Bug.

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

1-in-a billion bug.

```
private int hash(Key key)
{  return Math.abs(key.hashCode()) % M;  }
```

Correct.

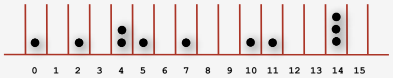
```
private int hash(Key key)
{  return (key.hashCode() & 0x7fffffff) % M;  }
```

- hash functions
- **separate chaining**
- linear probing
- applications

Helpful results from probability theory

Uniform hashing assumption. Each key is equally likely to hash to an integer between 0 and M-1.

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



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

Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

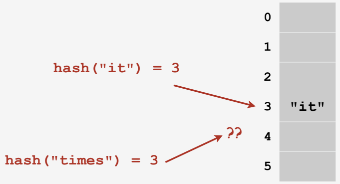
Load balancing. After M tosses, expect most loaded bin has $\Theta(\log M / \log \log M)$ balls.

Collisions

Collision. Two distinct keys hashing to same index.

- Birthday problem ⇒ can't avoid collisions unless you have a ridiculous amount (quadratic) of memory.
- Coupon collector + load balancing ⇒ collisions will be evenly distributed.

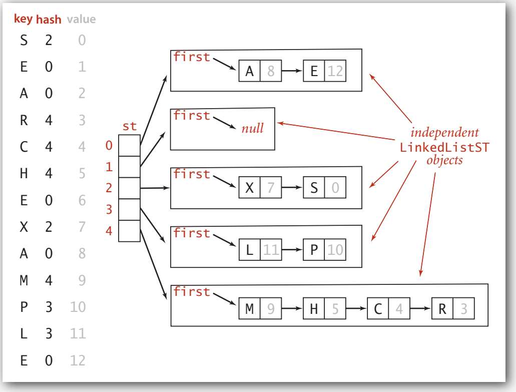
Challenge. Deal with collisions efficiently.



Separate chaining ST

Use an array of M < N linked lists. [H. P. Luhn, IBM 1953]

- 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	value
S	2	0
E	0	1
A	0	2
R	4	3
C	4	4
H	4	5
E	0	6
X	2	7
A	0	8
M	4	9
P	3	10
L	3	11
E	0	12

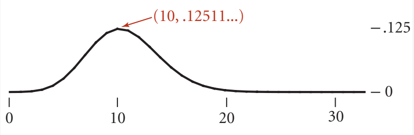
Separate chaining ST: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
   private int N;                // number of key-value pairs
   private int M;                // hash table size
   private LinkedListST[] st;    // array of STs
   public SCHashST()
   {  this(997);  }

   public SCHashST(int M)
   {  // Create M sequential-search-with-linked-list STs.
      this.M = M;
      st = new LinkedListST[M];
      for (int i = 0; i < M; i++)
         st[i] = new LinkedListST();
   }
   private int hash(Key key)
   {  return (key.hashCode() & 0x7fffffff) % M; }

   public Value get(Key key)
   {  return (Value) st[hash(key)].get(key);  }

   public void put(Key key, Value value)
   {  st[hash(key)].put(key, value);  }

   public Iterable<Key> keys()
   {  return st[i].keys();  }

}
```

Analysis of separate chaining

Proposition. Under uniform hashing assumption, probability that the number of keys in a list is within a constant factor of N/M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.



Binomial distribution $(N = 10^4, M = 10^3, \alpha = 10)$

Consequence. Number of compares for search/insert is proportional to N/M.

- M too large ⇒ too many empty chains.
- M too small ⇒ chains too long.
- Typical choice: M ~ N/5 ⇒ constant-time ops.

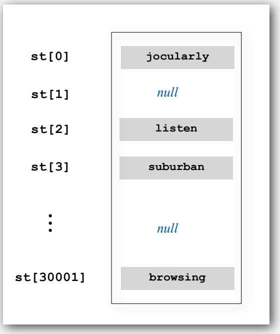
↑ M times faster than sequential search

- hash functions
- separate chaining
- **linear probing**
- applications

Collision resolution: open addressing

Open addressing. [Amdahl-Boehme-Rocherster-Samuel, IBM 1953]

When a new key collides, find next empty slot, and put it there.

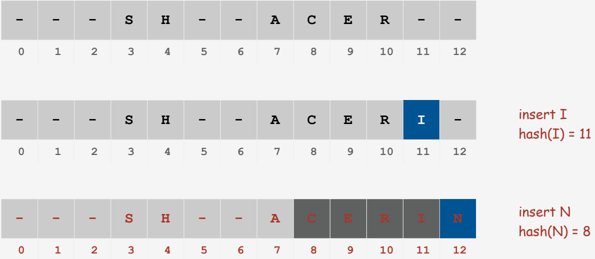


linear probing (M = 30001, N = 15000)

Linear probing

Use an array of size M > N.

- 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.



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

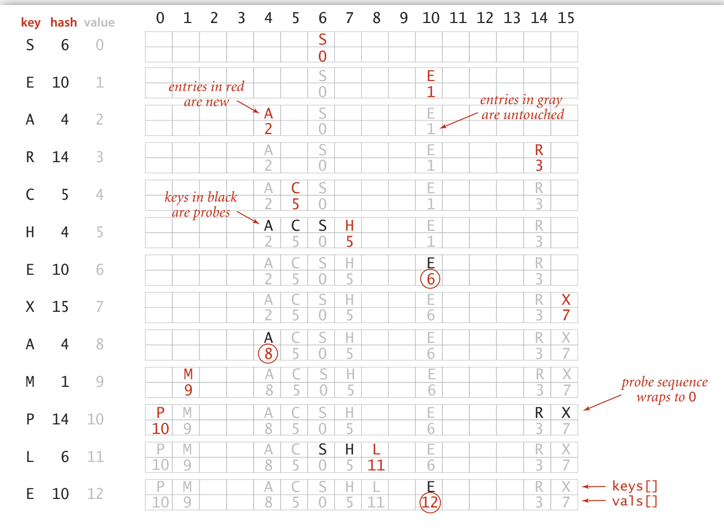
insert I
hash(I) = 11

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

insert N
hash(N) = 8

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

Linear probing: trace of standard indexing client



key	hash	value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
S	6	0							S 0									
E	10	1							S 0				E 1					
A	4	2					A 2		S 0				E 1					
R	14	3					A 2		S 0				E 1				R 3	
C	5	4					A 2	C 5	S 0				E 1				R 3	
H	4	5					A 2	C 5	S 0	H 5			E 1				R 3	
E	10	6					A 2	C 5	S 0	H 5			E 6				R 3	
X	15	7					A 2	C 5	S 0	H 5			E 6				R 3	X 7
A	4	8					A 8	C 5	S 0	H 5			E 6				R 3	X 7
M	1	9		M 9			A 8	C 5	S 0	H 5			E 6				R 3	X 7
P	14	10	P 10	M 9			A 8	C 5	S 0	H 5			E 6				R 3	X 7
L	6	11	P 10	M 9			A 8	C 5	S 0	H 5	L 11		E 6				R 3	X 7
E	10	12	P 10	M 9			A 8	C 5	S 0	H 5	L 11		E 12				R 3	X 7

Linear probing ST implementation

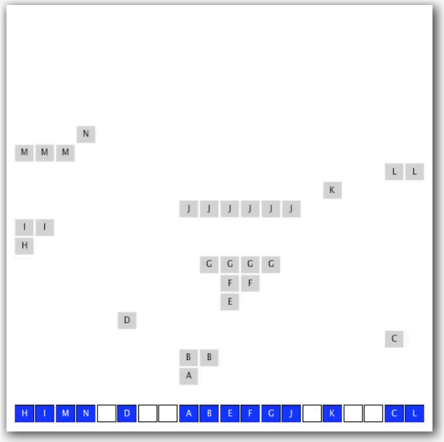
```
public class LinearProbingST<Key, Value>
{
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[]   keys = (Key[])   new Object[M];

   private 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;
   }
}
```

array doubling code omitted

Clustering

Cluster. A contiguous block of items.

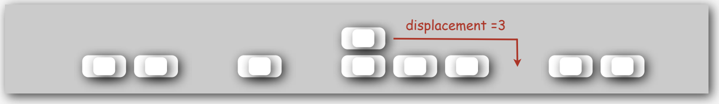
Observation. New keys likely to hash into middle of big clusters.



Knuth's parking problem

Model. 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, …

Q. What is mean displacement of a car?



Empty. With M/2 cars, mean displacement is ~ 3/2.

Full. With M cars, mean displacement is ~ $\sqrt{\pi M / 8}$

Analysis of linear probing

Proposition. Under uniform hashing assumption, the average number of probes in a hash table of size M that contains N = α M keys is:

$$\sim \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \qquad \sim \frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$$

search hit   search miss / insert

Pf. [Knuth 1962] A landmark in analysis of algorithms.

Parameters.

- M too large ⇒ too many empty array entries.
- M too small ⇒ search time blows up.
- Typical choice: α = N/M < 1/2 ⇒ constant-time ops.

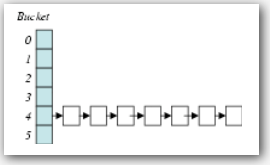
ST implementations: summary

implementation	guarantee			average case			ordered iteration?	operations on keys
	search	insert	delete	search hit	insert	delete		
sequential search (linked list)	N	N	N	N/2	N	N/2	no	`equals()`
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	`compareTo()`
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	`compareTo()`
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	`compareTo()`
hashing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	`equals()`

* under uniform hashing assumption

Algorithmic complexity attacks

Q. Is the uniform hashing assumption important in practice?
A. Obvious situations: aircraft control, nuclear reactor, pacemaker.
A. Surprising situations: denial-of-service attacks.



malicious adversary learns your hash function (e.g., by reading Java API) and causes a big pile-up in single slot 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.

Algorithmic complexity attack on Java

Goal. Find family of 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

key	hashCode()
"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!

Diversion: 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. $\underbrace{\text{MD4, MD5, SHA-0, SHA-1}}_{\text{known to be insecure}}$, 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.
Caveat. Too expensive for use in ST implementations.

Separate chaining vs. linear probing

Separate chaining.

- Easier to implement delete.
- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

Linear probing.

- Less wasted space.
- Better cache performance.

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. (separate chaining variant)

- Hash to two positions, put key in shorter of the two chains.
- Reduces average length of the longest chain to log log N.

Double hashing. (linear probing variant)

- Use linear probing, but skip a variable amount, not just 1 each time.
- Effectively eliminates clustering.
- Can allow table to become nearly full.

Hashing vs. balanced trees

Hashing.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops versus log N compares).
- Better system support in Java for strings (e.g., cached hash code).

Balanced trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- 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`.

- hash functions
- separate chaining
- linear probing
- applications

Set API

Mathematical set. A collection of distinct keys.

`public class SET<Key extends Comparable<Key>>`		
	`SET()`	*create an empty set*
`void`	`add(Key key)`	*add the key to the set*
`boolean`	`contains(Key key)`	*is the key in the set?*
`void`	`remove(Key key)`	*remove the key from the set*
`int`	`size()`	*return the number of keys in the set*
`Iterator<Key>`	`iterator()`	*iterator through keys in the set*

Q. How to implement?

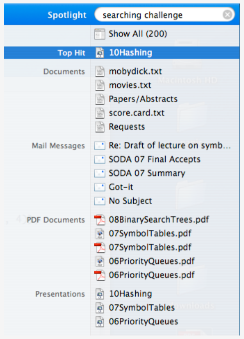
Searching challenge 5

Problem. Index for a PC or the web.

Assumptions. 1 billion++ words to index.

Which searching method to use?

- Hashing
- Red-black-trees
- Doesn't matter much.



Spotlight searching challenge

Show All (200)

Top Hit 10Hashing

Documents
mobydick.txt
movies.txt
Papers/Abstracts
score.card.txt
Requests

Mail Messages
Re: Draft of lecture on symb...
SODA 07 Final Accepts
SODA 07 Summary
Got-it
No Subject

PDF Documents
08BinarySearchTrees.pdf
07SymbolTables.pdf
07SymbolTables.pdf
06PriorityQueues.pdf
06PriorityQueues.pdf

Presentations
10Hashing
07SymbolTables
06PriorityQueues

Index for a PC or the web

Solution. Symbol table with:

- Key = query string.
- Value = set of pointers to files.

```
ST<String, SET<File>> st = new ST<String, SET<File>>();
for (File file : filesystem)
{
   In in = new In(file);
   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(file);
   }
}
```

build index

```
SET<File> files = st.get(query);
for (File file : files) ...
```

process lookup request

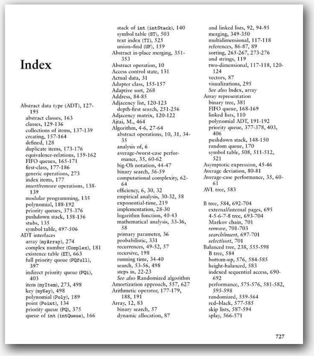
Searching challenge 6

Problem. Index for an e-book.

Assumptions. Book has 100,000+ words.

Which searching method to use?

1. Hashing
2. Red-black-tree
3. Doesn't matter much.



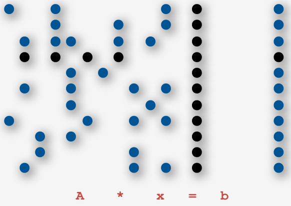
Index

727

Searching challenge 2

Problem. Sparse matrix-vector multiplication.

Assumptions. Matrix dimension is 10,000; average nonzeros per row ~ 10.



Matrix-vector multiplication (standard implementation)

$$\overset{\texttt{a[][]}}{\begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix}} \overset{\texttt{x[]}}{\begin{bmatrix} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{bmatrix}} = \overset{\texttt{b[]}}{\begin{bmatrix} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{bmatrix}}$$

```
...
double[][] a = new double[N][N];
double[] x = new double[N];
double[] b = new double[N];
...
// Initialize a[][] and x[].
...
for (int i = 0; i < N; i++)
{
   sum = 0.0;
   for (int j = 0; j < N; j++)
      sum += a[i][j]*x[j];
   b[i] = sum;
}
```

nested loops
N^2 running time

Vector representations

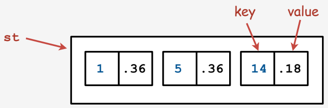
1D array (standard) representation.

- Constant time access to elements.
- Space proportional to N.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	.36	0	0	0	.36	0	0	0	0	0	0	0	0	.18	0	0	0	0	0

Symbol table representation.

- key = index, value = entry
- Efficient iterator.
- Space proportional to number of nonzeros.



Sparse vector data type

```
public class SparseVector
{
   private HashST<Integer, Double> v;
   public SparseVector()
    {  v = new HashST<Integer, Double>();  }

   public void put(int i, double x)
    {  v.put(i, x);  }

   public double get(int i)
   {
      if (!v.contains(i)) return 0.0;
      else return v.get(i);
   }
   public Iterable<Integer> indices()
   {  return v.keys();  }

   public double dot(double[] that)
   {
       double sum = 0.0;
       for (int i : v.indices())
           sum += that[i]*this.get(i);
       return sum;
   }
}
```

`HashST` because order not important

empty ST represents all 0s vector

`a[i] = value`

`return a[i]`

dot product is constant time for sparse vectors

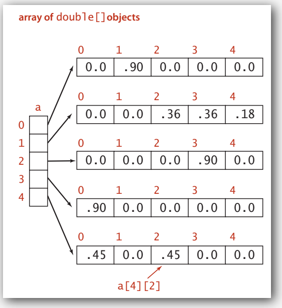
Matrix representations

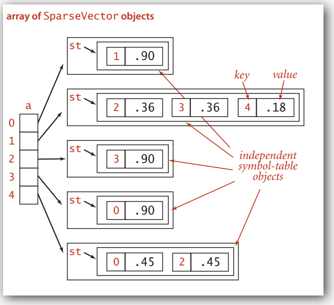
2D array (standard) representation: Each row of matrix is an array.

- Constant time access to elements.
- Space proportional to N^2.

Sparse representation: Each row of matrix is a sparse vector.

- Efficient access to elements.
- Space proportional to number of nonzeros (plus N).





Sparse matrix-vector multiplication

$$\overset{\texttt{a[][]}}{\begin{bmatrix} 0 & .90 & 0 & 0 & 0 \\ 0 & 0 & .36 & .36 & .18 \\ 0 & 0 & 0 & .90 & 0 \\ .90 & 0 & 0 & 0 & 0 \\ .47 & 0 & .47 & 0 & 0 \end{bmatrix}} \overset{\texttt{x[]}}{\begin{bmatrix} .05 \\ .04 \\ .36 \\ .37 \\ .19 \end{bmatrix}} = \overset{\texttt{b[]}}{\begin{bmatrix} .036 \\ .297 \\ .333 \\ .045 \\ .1927 \end{bmatrix}}$$

```
..
SparseVector[] a;
a = new SparseVector[N];
double[] x = new double[N];
double[] b = new double[N];
...
// Initialize a[] and x[].
...
for (int i = 0; i < N; i++)
   b[i] = a[i].dot(x);
```

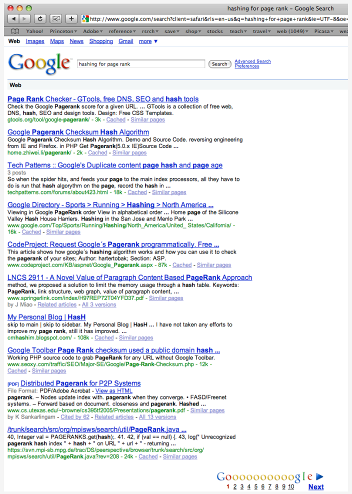
one loop
linear running time
for sparse matrix

Searching challenge 7

Problem. Rank pages on the web.

Assumptions.

- Matrix-vector multiply
- 10 billion+ rows
- sparse



Page Rank Checker - GTools, free DNS, SEO and hash tools
Check the Google **Pagerank** score for a given URL. ... GTools is a collection of free web, DNS, **hash**, SEO and design tools. Design: Free CSS Templates.
gtools.org/tool/google-**pagerank**/ - 3k - Cached - Similar pages

Google Pagerank Checksum Hash Algorithm
Google **Pagerank** Checksum **Hash** Algorithm. Demo and Source Code. reversing engineering from IE and Firefox. in PHP Get **Pagerank**(5.0.x IE)Source Code ...
home.zhiwei.li/**pagerank**/ - 2k - Cached - Similar pages

Tech Patterns :: Google's Duplicate content page hash and page age
3 posts
So when the spider hits, and feeds your **page** to the main index processors, all they have to do is run that **hash** algorythm on the **page**, record the **hash** in ...
techpatterns.com/forums/about423.html - 18k - Cached - Similar pages

Google Directory - Sports > Running > Hashing > North America ...
Viewing in Google **PageRank** order View in alphabetical order ... Home **page** of the Silicone Valley **Hash** House Harriers. **Hashing** in the San Jose and Menlo Park ...
www.google.com/Top/Sports/Running/**Hashing**/North_America/United_ States/California/ - 16k - Cached - Similar pages

CodeProject: Request Google´s Pagerank programmatically. Free ...
This article shows how google´s **hashing** algorithm works and how you can use it to check the **pagerank** of your sites; Author: hartertobak; Section: ASP.
www.codeproject.com/KB/aspnet/Google_**Pagerank**.aspx - 87k - Cached - Similar pages

LNCS 2911 - A Novel Value of Paragraph Content Based PageRank Approach
method, we proposed a solution to limit the memory usage through a **hash** table. Keywords: **PageRank**, link structure, web graph, value of paragraph content, ...
www.springerlink.com/index/H97REP72T04YFD37.pdf - Similar pages
by J Miao - Related articles - All 3 versions

My Personal Blog | HasH
skip to main | skip to sidebar. My Personal Blog | **HasH** ... I have not taken any efforts to improve my **page rank**, still it has improved. ...
cm**hash**im.blogspot.com/ - 108k - Cached - Similar pages

Google Toolbar Page Rank checksum used a public domain hash ...
Working PHP source code to grab **PageRank** for any URL without Google Toolbar.
www.seoxy.com/traffic/SEO/Major-SE/Google/**Page-Rank**-Checksum.php - 12k - Cached - Similar pages

[PDF] **Distributed Pagerank for P2P Systems**
File Format: PDF/Adobe Acrobat - View as HTML
pagerank. – Nodes update index with. **pagerank** when they converge. • FASD/Freenet systems. – Forward based on document. closeness and **pagerank**. **Hashed** ...
www.cs.utexas.edu/~browne/cs395tf2005/Presentations/**pagerank**.pdf - Similar pages
by K Sankarlingam - Cited by 62 - Related articles - All 13 versions

/trunk/search/src/org/mpisws/search/util/PageRank.java ...
40, Integer val = PAGERANKS.get(**hash**);. 41. 42, if (val == null) {. 43, log(" Unrecognized **pagerank hash** index " + **hash** + " on URL " + url + " - returning ...
https://svn.mpi-sb.mpg.de/trac/DS/peerspective/browser/trunk/search/src/org/mpisws/search/util/**PageRank**.java?rev=208 - 24k - Cached - Similar pages

Which "searching" method to use to access array values?

1. Standard 2D array representation
2. Symbol table
3. Doesn't matter much.