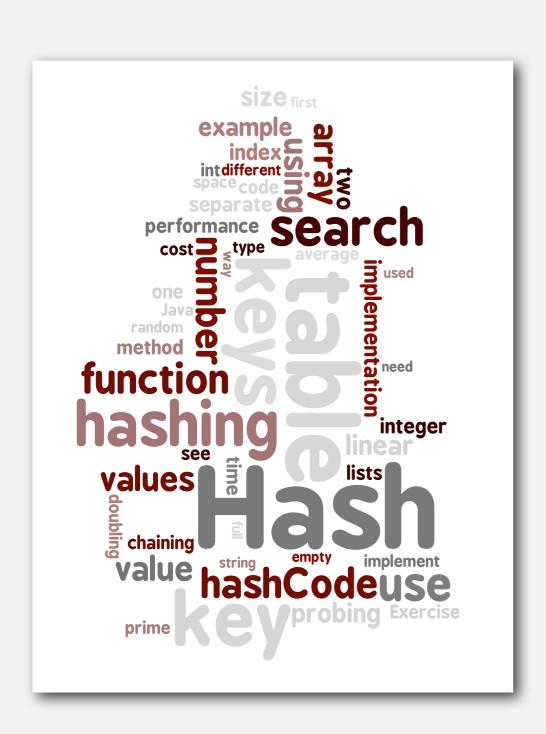
Hash Tables



- hash functions
- separate chaining
- linear probing
- applications

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	operations
	search	insert	delete	search hit	insert	delete	iteration?	on keys
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.

1
hash("it") = 3
2
"it"

Issues.

133ue3.

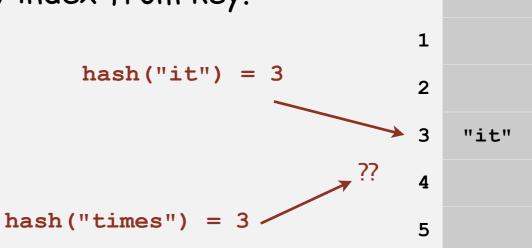
- Computing the hash function.
- Equality test: Method for checking whether two keys are equal.

5

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.



0

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.
- Space and time limitations: hashing (the real world).

hash functions

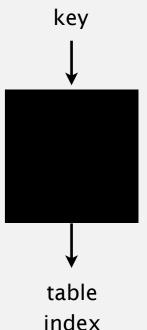
- > separate chaining
- linear probing
- applications

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.

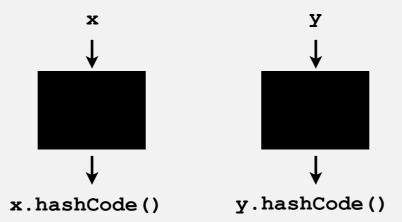
Practical challenge. Need different approach for each key type.

Java's hash code conventions

All Java classes inherit a method hashcode (), which returns a 32-bit int.

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

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



Default implementation. Memory address of x.

Trivial (but poor) implementation. Always return 17.

Customized implementations. Integer, Double, String, File, URL, Date, ...

User-defined types. Users are on their own.

Implementing hash code: integers, booleans, and doubles

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

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

```
public final class Boolean
{
    private final boolean value;
    ...

public int hashCode()
    {
        if (value) return 1231;
        else return 1237;
     }
}
```

```
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;
    }
}</pre>
```

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

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

War story: String hashing in Java

String hashCode() 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;
}</pre>
```

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/12type/index.html
```

Implementing hash code: user-defined types

```
public final class Transaction
   private final long who;
   private final Date when;
   private final String where;
   public Transaction(long who, Date when, String where)
   { /* as before */ }
   public boolean equals(Object y)
   {    /* as before */ }
   public int hashCode()
                                 nonzero constant
                                                                   for primitive types,
      int hash = 17;
                                                                   USE hashCode()
      hash = 31*hash + ((Long) who).hashCode(); 

                                                                   of wrapper type
      hash = 31*hash + when.hashCode();
      hash = 31*hash + where.hashCode(); 
                                                                   for reference types,
      return hash;
                                                                   USE hashCode()
                        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 wrapper type hashcode().
- If field is an array, apply to each element. ← or use Arrays. deepHashCode()
- If field is a reference type, use hashcode(). ← applies 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.

Modular hashing

Hash code. An int between -231 and 231-1.

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

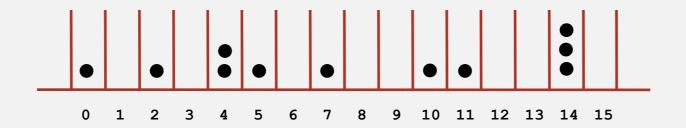
typically a prime or power of 2

```
private int hash(Key key)
     return key.hashCode() % M; }
bug
 private int hash(Key key)
     return Math.abs(key.hashCode()) % M; }
1-in-a-billion bug
                    hashCode() of "polygenelubricants" is -231
 private int hash(Key key)
     return (key.hashCode() & 0x7ffffffff) % M; }
correct
```

Uniform hashing assumption

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.

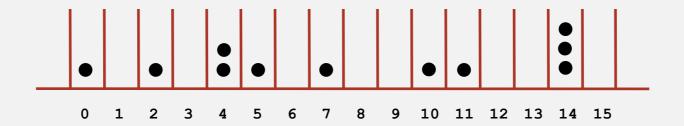
Coupon collector. Expect every bin has ≥ 1 ball after $\sim M \ln M$ tosses.

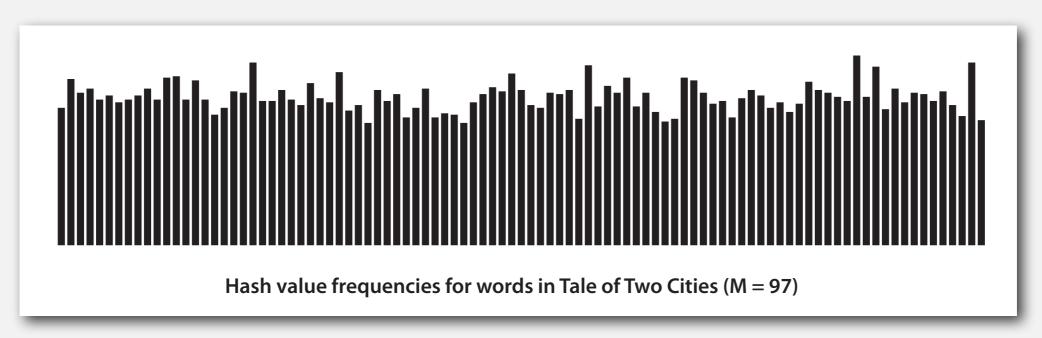
Load balancing. After M tosses, expect most loaded bin has $\Theta\left(\log M/\log\log M\right)$ balls.

Uniform hashing assumption

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.





Java's String data uniformly distribute the keys of Tale of Two Cities

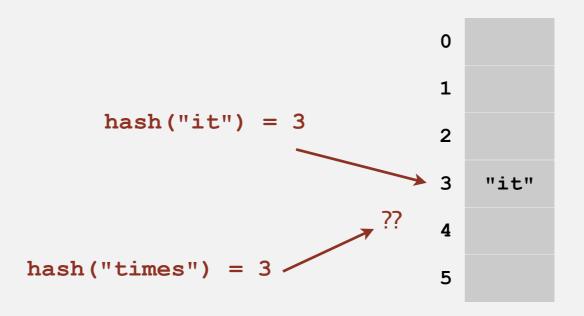
- hash functions
- separate chaining
- linear probing
- applications

Collisions

Collision. Two distinct keys hashing to same index.

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

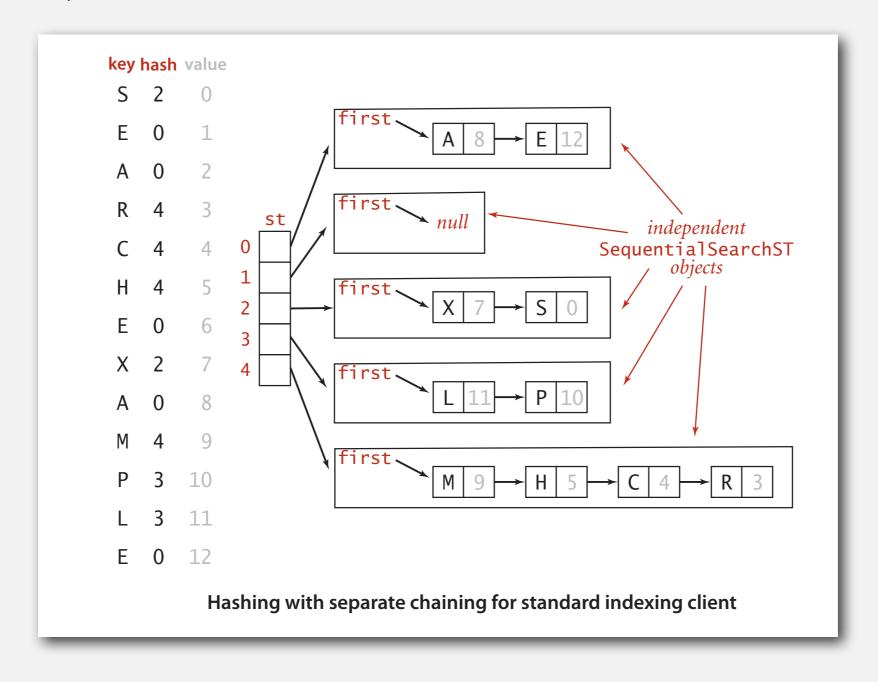
Challenge. Deal with collisions efficiently.



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 ith chain.



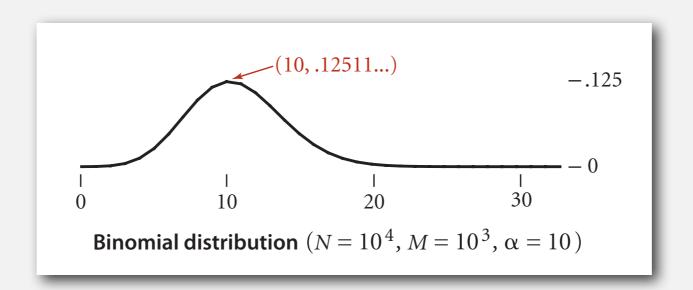
Separate chaining ST: Java implementation

```
public class SeparateChainingHashST<Key, Value>
  private int N;  // number of key-value pairs
  private int M;  // hash table size
  private SequentialSearchST<Key, Value> [] st; // array of STs
  public SeparateChainingHashST()
                                            array doubling and halving code omitted
   { this(997); }
  public SeparateChainingHashST(int M)
      this.M = M;
      st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[M];
      for (int i = 0; i < M; i++)
         st[i] = new SequentialSearchST<Key, Value>();
   private int hash(Key key)
      return (key.hashCode() & 0x7fffffff) % M; }
   public Value get(Key key)
     return st[hash(key)].get(key); }
  public void put(Key key, Value val)
      st[hash(key)].put(key, val); }
```

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.



equals() and hashCode()

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

- M too large \Rightarrow too many empty chains.
- M too small \Rightarrow chains too long.
- Typical choice: $M \sim N/5 \Rightarrow \text{constant-time ops.}$

M times faster than sequential search

ST implementations: summary

implementation	guarantee			average case			ordered	operations
	search	insert	delete	search hit	insert	delete	iteration?	on keys
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()
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	equals()

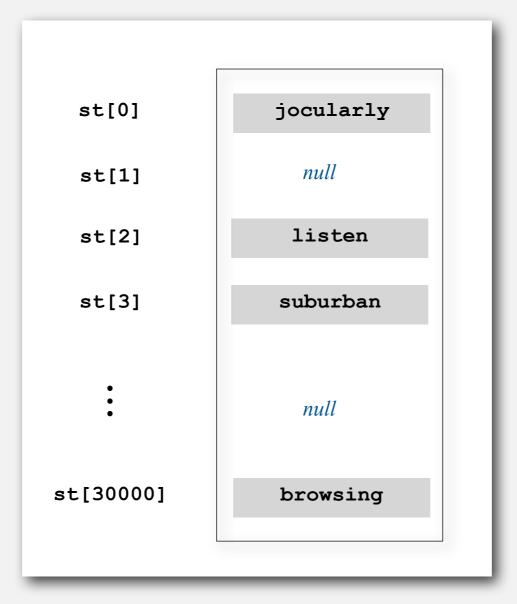
^{*} under uniform hashing assumption

- 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.

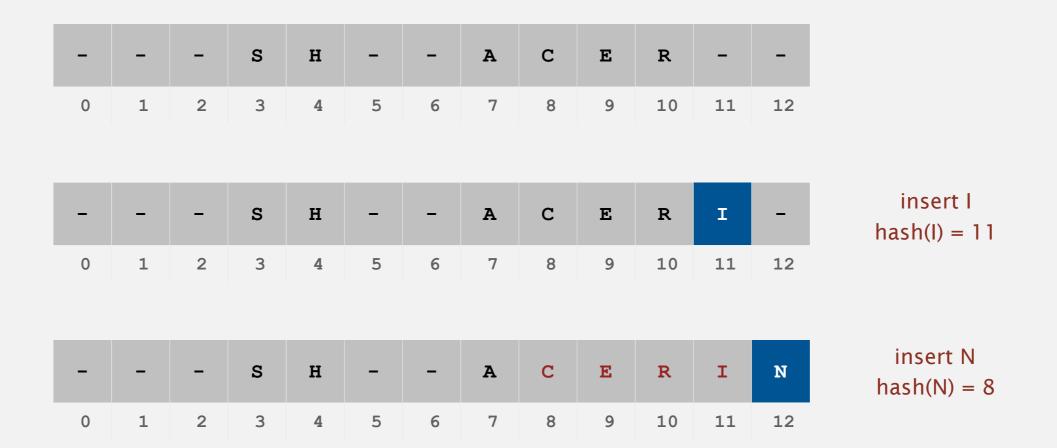


linear probing (M = 30001, N = 15000)

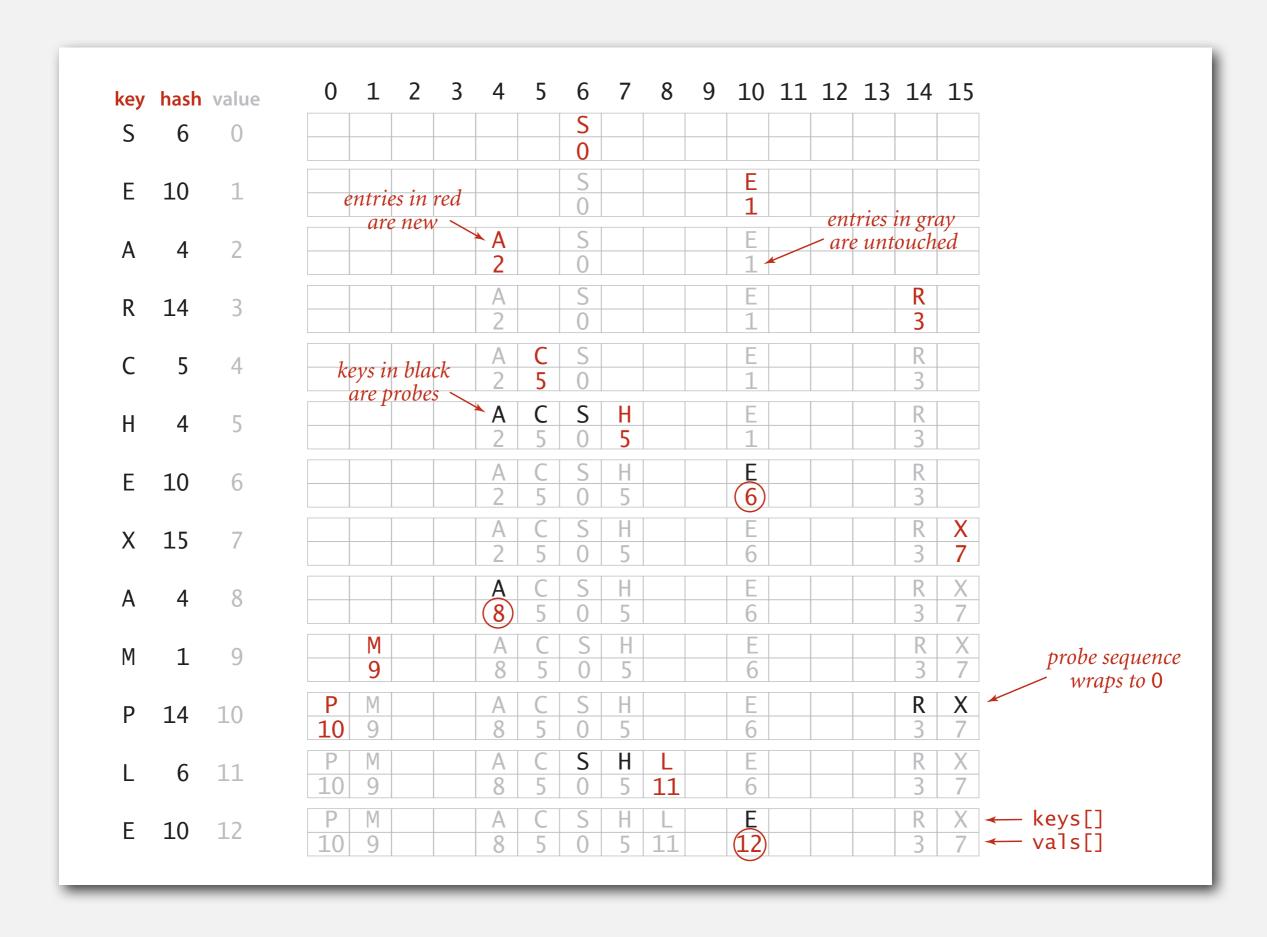
Linear probing

Use an array of size M > N.

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



Linear probing: trace of standard indexing client



Linear probing ST implementation

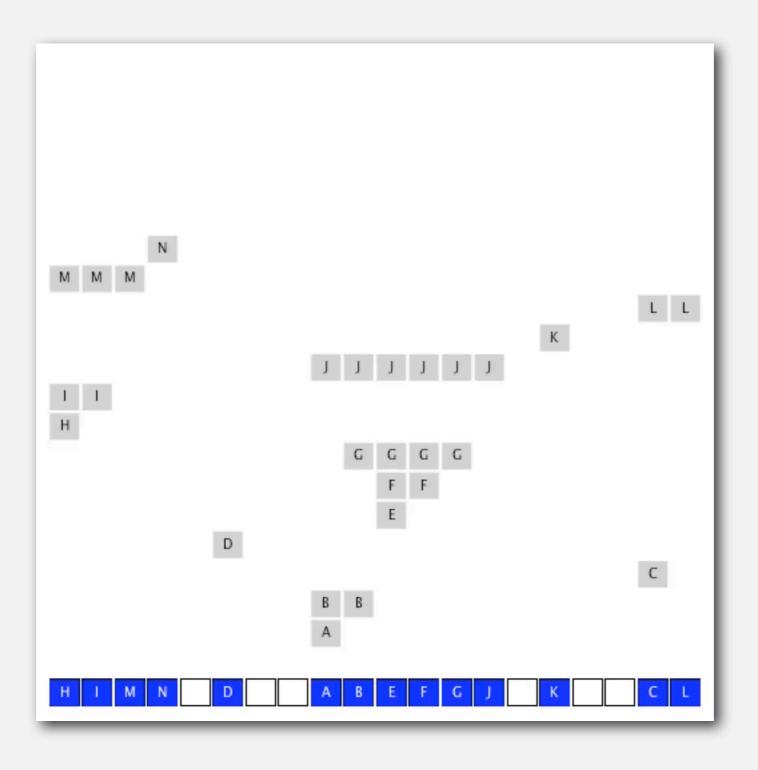
```
public class LinearProbingHashST<Key, Value>
   private int M = 30001;
   private Value[] vals = (Value[]) new Object[M];
   private Key[] keys = (Key[]) new Object[M];
   private int hash(Key key) { /* as before */ }
   public void put(Key key, Value val)
      int i;
      for (i = hash(key); keys[i] != null; i = (i+1) % M)
         if (keys[i].equals(key))
             break;
      keys[i] = key;
      vals[i] = val;
   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 and halving 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, etc.

Q. What is mean displacement of a car?



Half-full. With M/2 cars, mean displacement is $\sim 3/2$.

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

Analysis of linear probing

Proposition. Under uniform hashing assumption, the average number of probes in a hash table of size M that contains $N = \alpha M$ 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 \Rightarrow too many empty array entries.
- M too small \Rightarrow search time blows up.
- Typical choice: $\alpha = N/M \sim \frac{1}{2}$.

probes for search hit is about 3/2
probes for search miss is about 5/2

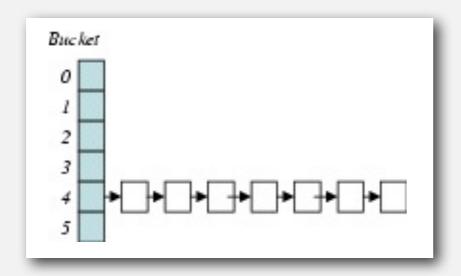
ST implementations: summary

implementation	guarantee			average case			ordered	operations
implementation	search	insert	delete	search hit	insert	delete	iteration?	on keys
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()
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	equals()
linear probing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	equals()

^{*} under uniform hashing assumption

War story: 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 two keys that hash to same value).

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

known to be 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. 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 expected 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.
- Difficult to implement delete.

Hashing vs. balanced search 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 search trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement compare To () correctly than equals () and hashcode ().

Java system includes both.

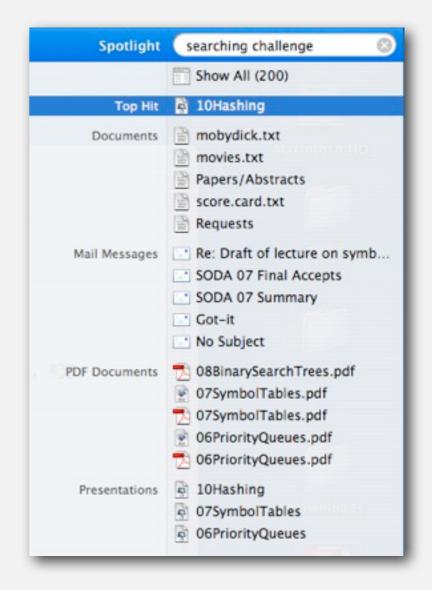
- Red-black trees: java.util.TreeMap, java.util.TreeSet.
- Hashing: java.util.HashMap, java.util.IdentityHashMap.

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.



Problem. Index for a PC or the web.

Assumptions. 1 billion++ words to index.

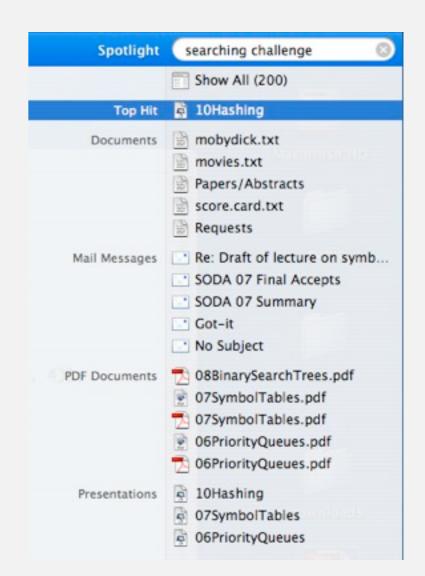
Which searching method to use?

- Hashing
- Red-black-trees ← too much space
- Doesn't matter much.

Solution. Symbol table with:

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





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

Address, 84-85 Abstract data type (ADT), 127abstract classes, 163 Ajtai, M., 464 classes, 129-136 collections of items, 137-139 creating, 157-164 defined, 128 analysis of, 6 duplicate items, 173-176 equivalence-relations, 159-162 FIFO queues, 165-171 first-class, 177-186 generic operations, 273 index items, 177 insertlremove operations, 138-139 modular programming, 135 polynomial, 188-192 priority queues, 375-376 pushdown stack, 138-156 stubs, 135 symbol table, 497-506 ADT interfaces array (myArray), 274 complex number (Complex), 181 existence table (ET), 663 recursive, 198 full priority queue (PQful1), indirect priority queue (PQ1), item (nyIten), 273, 498 key (nyKey), 498 polynomial (Poly), 189 Array, 12, 83 point (Point), 134 priority queue (PQ), 375 queue of int (intQueue), 166

stack of int (intStack), 140 symbol table (8T), 503 text index (TI), 525 union-find (UF), 159 Abstract in-place merging, 351-Abstract operation, 10 Access control state, 131 124 vectors, 87 Acreal data, 31 Adapter class, 155-157 Adaptive sort, 268 Adjacency list, 120-123 depth-first search, 251-256 Adjacency matrix, 120-122 Algorithm, 4-6, 27-64 406 abstract operations, 10, 31, 34average-/weest-case perfeemance, 35, 60-62 big-Oh notation, 44-47 binary search, 56-59 computational complexity, 62-AVL tree, 583 efficiency, 6, 30, 32 empirical analysis, 30-32, 58 exponential-time, 219 implementation, 28-30 logarithm function, 40-43 mathematical analysis, 33-36, primary parameter, 36 probabilistic, 331 recurrences, 49-52, 57 B tree, 584 running time, 34-40 search, 53-56, 498 steps in, 22-23 See also Randomized algorithm Amortization approach, 557, 627 Arithmetic operator, 177-179, 188, 191 595-598 binary search, 57 dynamic allocation, 87

and linked lists, 92, 94-95 merging, 349-350 multidimensional, 117-118 references, 86-87, 89 sorting, 265-267, 273-276 and strings, 119 two-dimensional, 117-118, 120visualizations, 295 See also Index, array Array representation binary tree, 381 HFO queue, 168-169 linked lists, 110 polynomial ADT, 191-192 priority queue, 377-378, 403, pushdown stack, 148-150 random queue, 170 symbol table, 508, 511-512, Asymptotic expression, 45-46 Average deviation, 80-81 Average-case performance, 35, 60-B tree, 584, 692-704 external/internal pages, 695 4-5-6-7-8 tree, 693-704 Markov chain, 701 zemore, 701-703 search/insert, 697-701 select/sort, 701 Balanced tree, 238, 555-598 bottom-up, 576, 584-585 height-balanced, 583 indexed sequential access, 690performance, 575-576, 581-582, randomized, 559-564 red-black, 577-585 skip lists, 587-594 splay, 566-571

727

Problem. Index for an e-book.

Assumptions. Book has 100,000+ words.

Which searching method to use?

- 1. Hashing
- ✓ 2. Red-black-tree need ordered iteration
 - 3. Doesn't matter much.

Solution. Symbol table with:

- Key = index term.
- Value = ordered set of pages on which term appears.

stack of int (intStack), 140 and linked lists, 92, 94-95 symbol table (8T), 503 merging, 349-350 text index (TI), 525 multidimensional, 117-118 union-find (UF), 159 references, 86-87, 89 Abstract in-place merging, 351sorting, 265-267, 273-276 and strings, 119 Index two-dimensional, 117-118, 120-Abstract operation, 10 Access control state, 131 124 vectors, 87 Acreal data, 31 Adapter class, 155-157 visualizations, 295 Adaptive sort, 268 See also Index, array Address, 84-85 Array representation Adjacency list, 120-123 binary tree, 381 Abstract data type (ADT), 127depth-first search, 251-256 HFO queue, 168-169 Adjacency matrix, 120-122 linked lists, 110 abstract classes, 163 polynomial ADT, 191-192 Ajtai, M., 464 classes, 129-136 Algorithm, 4-6, 27-64 priority queue, 377-378, 403, collections of items, 137-139 406 abstract operations, 10, 31, 34creating, 157-164 pushdown stack, 148-150 defined, 128 random queue, 170 analysis of, 6 duplicate items, 173-176 symbol table, 508, 511-512, average-/weest-case perforequivalence-relations, 159-162 mance, 35, 60-62 FIFO queues, 165-171 Asymptotic expression, 45-46 big-Oh notation, 44-47 first-class, 177-186 Average deviation, 80-81 binary search, 56-59 generic operations, 273 Average-case performance, 35, 60computational complexity, 62index items, 177 insert/remove operations, 138-AVL tree, 583 efficiency, 6, 30, 32 139 empirical analysis, 30-32, 58 modular programming, 135 exponential-time, 219 polynomial, 188-192 B tree, 584, 692-704 priority queues, 375-376 implementation, 28-30 external/internal pages, 695 logarithm function, 40-43 4-5-6-7-8 tree, 693-704 pushdown stack, 138-156 mathematical analysis, 33-36, Markov chain, 701 stubs, 135 symbol table, 497-506 zewoev, 701-703 primary parameter, 36 search/insert, 697-701 ADT interfaces probabilistic, 331 array (myArray), 274 select/sort, 701 recurrences, 49-52, 57 Balanced tree, 238, 555-598 complex number (Complex), 181 existence table (ET), 663 recursive, 198 B tree, 584 bottom-up, 576, 584-585 running time, 34-40 full priority queue (PQfull), search, 53-56, 498 height-balanced, 583 indirect priority queue (PQs), steps in, 22-23 indexed sequential access, 690-See also Randomized algorithm performance, 575-576, 581-582, item (nyIten), 273, 498 Amortization approach, 557, 627 Arithmetic operator, 177-179, 188, 191 595-598 key (nyKey), 498 randomized, 559-564 polynomial (Poly), 189 red-black, 577-585 Array, 12, 83 point (Point), 134 binary search, 57 skip lists, 587-594 priority queue (PQ), 375 queue of int (intQueue), 166 dynamic allocation, 87 splay, 566-571 727