

3.4 HASH TABLES

ST implementations: summary



- ▶ hash functions
- ▶ separate chaining
- ▶ linear probing

implementation	worst-case cost (after N inserts)			average-case cost (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	<code>compareTo()</code>
red-black BST	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	<code>compareTo()</code>

Q. Can we do better?

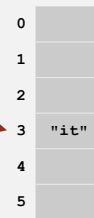
A. Yes, but with different access to the data.

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.

`hash("it") = 3`



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.

`hash("it") = 3`

`hash("times") = 3`



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

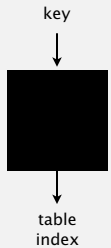
5

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.

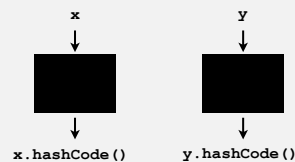
6

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`.

Legal (but poor) implementation. Always return 17.

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

User-defined types. Users are on their own.

7

Implementing hash code: integers, booleans, and doubles

Java library implementations

```
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

8

Implementing hash code: strings

Java library implementation

```
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;
    }
}
```

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

*i*th character of s

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

Ex. `String s = "call";`
`int code = s.hashCode();` ← $3045982 = 99 \cdot 31^3 + 97 \cdot 31^2 + 108 \cdot 31^1 + 108 \cdot 31^0$
 $= 108 + 31 \cdot (108 + 31 \cdot (97 + 31 \cdot (99)))$
 (Horner's method)

9

Implementing hash code: strings

Performance optimization.

- Cache the hash value in an instance variable.
- Return cached value.

```
public final class String
{
    private int hash = 0; ← cache of hash code
    private final char[] s;
    ...

    public int hashCode()
    {
        int h = hash; ← return cached value
        if (h != 0) return h;
        for (int i = 0; i < length(); i++)
            h = s[i] + (31 * hash);
        hash = h; ← store cache of hash code
        return h;
    }
}
```

10

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;
}
```

- 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
↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
```

11

Implementing hash code: user-defined types

```
public final class Transaction implements Comparable<Transaction>
{
    private final String who;
    private final Date when;
    private final double amount;

    public Transaction(String who, Date when, double amount)
    { /* as before */ }

    ...

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

    public int hashCode()
    {
        int hash = 17; ← nonzero constant
        hash = 31*hash + who.hashCode(); ← for reference types, use hashCode()
        hash = 31*hash + when.hashCode();
        hash = 31*hash + ((Double) amount).hashCode(); ← for primitive types, use hashCode() of wrapper type
        return hash;
    }
}
```

typically a small prime

12

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 null, return 0.
- If field is a reference type, use `hashCode()`. ← applies rule recursively
- If field is an array, apply to each entry. ← or use `Arrays.deepHashCode()`

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

In theory. Keys are bitstring; "universal" hash functions exist.

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

13

Modular hashing

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

```
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 -2^{31}

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

correct

14

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

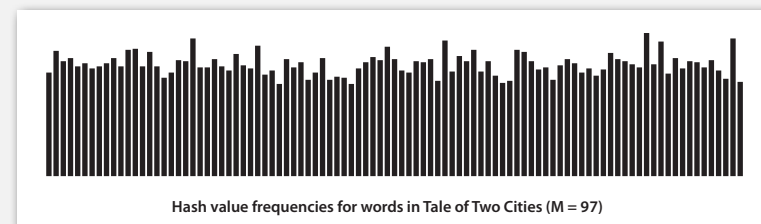
$\Theta(\log M / \log \log M)$ balls.

15

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.



Hash value frequencies for words in Tale of Two Cities ($M = 97$)

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

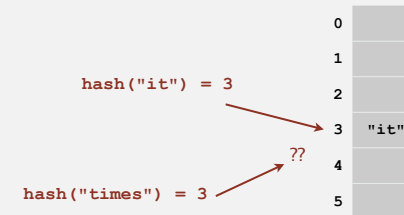
16

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.



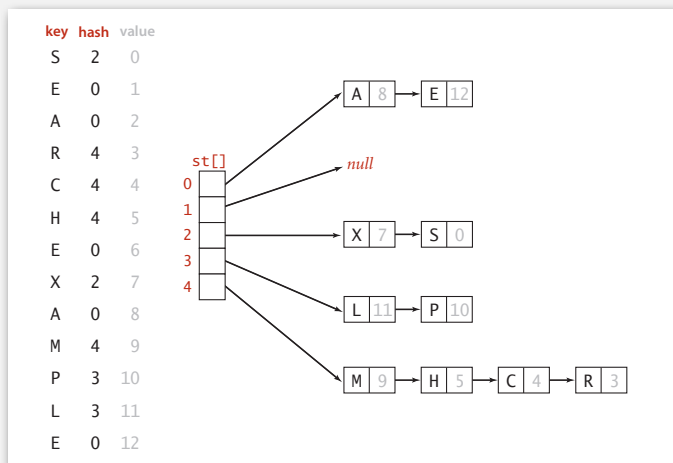
17

18

Separate chaining symbol table

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: need to search only i^{th} chain.



19

Separate chaining ST: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
    private int M = 97; // number of chains
    private Node[] st = new Node[M]; // array of chains

    private static class Node
    {
        private Object key;
        private Object val;
        private Node next;
        ...
    }

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

    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;
    }
}
```

array doubling
and halving
code omitted

20

Separate chaining ST: Java implementation

```
public class SeparateChainingHashST<Key, Value>
{
    private int M = 97;           // number of chains
    private Node[] st = new Node[M]; // array of chains

    private static class Node
    {
        private Object key;
        private Object val;
        private Node next;
        ...
    }

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

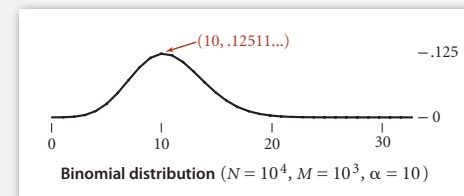
    public void put(Key key, Value val) {
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)
            if (key.equals(x.key)) { x.val = val; return; }
        st[i] = new Node(key, val, st[i]);
    }
}
```

21

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.



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: $M \sim N/5 \Rightarrow$ constant-time ops.

↑
M times faster than
sequential search

22

ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	<code>equals()</code>
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	<code>compareTo()</code>
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	<code>compareTo()</code>
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	<code>compareTo()</code>
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	<code>equals()</code>

* under uniform hashing assumption

23

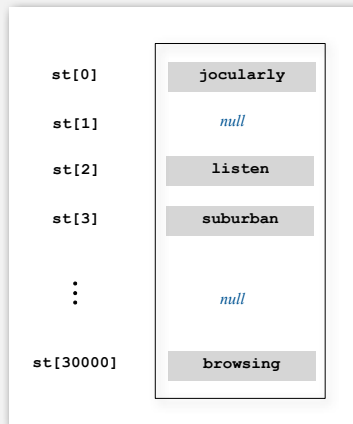
- hash functions
- separate chaining
- linear probing

24

Collision resolution: open addressing

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

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



linear probing ($M = 30001$, $N = 15000$)

25

Linear probing demo

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.

Note. Array size M must be greater than N .



$M = 16$

26

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;
    }
}
```

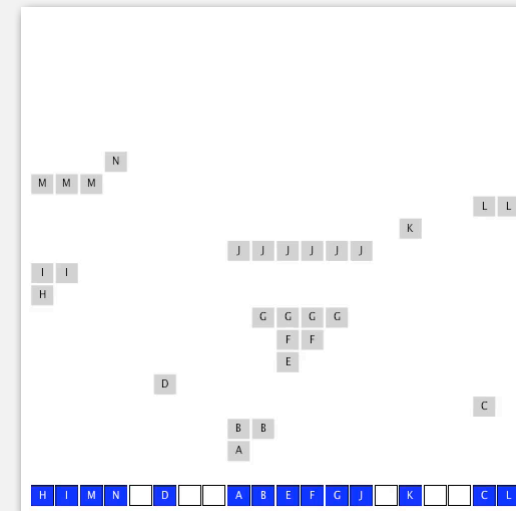
array doubling
and halving
code omitted

27

Clustering

Cluster. A contiguous block of items.

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



28

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 $\sim \sqrt{\pi M/8}$

29

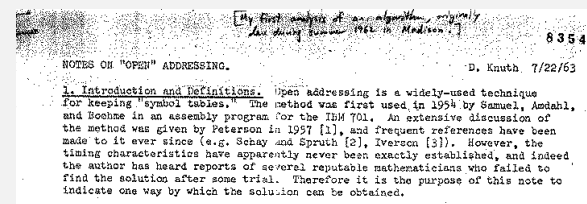
Analysis of linear probing

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

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

search hit search miss / insert

Pf.



Parameters.

- M too large \Rightarrow too many empty array entries.
- M too small \Rightarrow search time blows up.
- Typical choice: $\alpha = N/M \sim 1/2$. \leftarrow # probes for search hit is about 3/2
probes for search miss is about 5/2

30

ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	$N/2$	N	$N/2$	no	<code>equals()</code>
binary search (ordered array)	$\lg N$	N	N	$\lg N$	$N/2$	$N/2$	yes	<code>compareTo()</code>
BST	N	N	N	$1.38 \lg N$	$1.38 \lg N$?	yes	<code>compareTo()</code>
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	<code>compareTo()</code>
separate chaining	$\lg N^*$	$\lg N^*$	$\lg N^*$	3.5^*	3.5^*	3.5^*	no	<code>equals()</code>
linear probing	$\lg N^*$	$\lg N^*$	$\lg N^*$	3.5^*	3.5^*	3.5^*	no	<code>equals()</code>

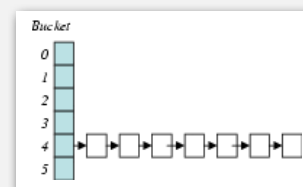
* under uniform hashing assumption

31

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.

32

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 ()	key	hashCode ()	key	hashCode ()
"Aa"	2112	"AaAaAaAa"	-540425984	"BBAaAaAa"	-540425984
"BB"	2112	"AaAaAaBB"	-540425984	"BBAaAaBB"	-540425984
		"AaAaBBAa"	-540425984	"BBAaBBAa"	-540425984
		"AaAaBBBB"	-540425984	"BBAaBBBB"	-540425984
		"AaBBAaAa"	-540425984	"BBBBAaAa"	-540425984
		"AaBBAaBB"	-540425984	"BBBBAaBB"	-540425984
		"AaBBBBAa"	-540425984	"BBBBBAaA"	-540425984
		"AaBBBBBB"	-540425984	"BBBBBAaBB"	-540425984
				"BBBBBBaA"	-540425984
				"BBBBBBBB"	-540425984

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

33

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.

34

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.

Q. How to delete?

Q. How to resize?

35

Hashing: variations on the theme

Many improved versions have been studied.

Two-probe hashing. (separate-chaining variant)

- Hash to two positions, insert 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.
- More difficult to implement delete.

Cuckoo hashing. (linear-probing variant)

- Hash key to two positions; insert key into either position; if occupied, reinsert displaced key into its alternative position (and recur).
- Constant worst case time for search.



36

Hash tables.

- 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 `compareTo()` correctly than `equals()` and `hashCode()`.

Java system includes both.

- Red-black BSTs: `java.util.TreeMap`, `java.util.TreeSet`.
- Hash tables: `java.util.HashMap`, `java.util.IdentityHashMap`.

37

3.5 SYMBOL TABLE APPLICATIONS



- ▶ sets
- ▶ dictionary clients
- ▶ indexing clients
- ▶ sparse vectors

Algorithms, 4th Edition · Robert Sedgwick and Kevin Wayne · Copyright © 2002–2012 · March 7, 2012 5:02:56 AM

- ▶ sets
- ▶ dictionary clients
- ▶ indexing clients
- ▶ sparse vectors

2

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?

3

Exception filter

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

```
% more list.txt
was it the of

% java WhiteList list.txt < tinyTale.txt
it was the of it was the of
it was the of it was the of
it was the of it was the of
it was the of it was the of
it was the of it was the of

% java BlackList list.txt < tinyTale.txt
best times worst times
age wisdom age foolishness
epoch belief epoch incredulity
season light season darkness
spring hope winter despair
```

← list of exceptional words

4

Exception filter applications

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

application	purpose	key	in list
spell checker	identify misspelled words	word	dictionary words
browser	mark visited pages	URL	visited pages
parental controls	block sites	URL	bad sites
chess	detect draw	board	positions
spam filter	eliminate spam	IP address	spam addresses
credit cards	check for stolen cards	number	stolen cards

5

Exception filter: Java implementation

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

```
public class WhiteList
{
    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 empty set of strings

← read in whitelist

← print words not in list

6

Exception filter: Java implementation

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

```
public class BlackList
{
    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 empty set of strings

← read in whitelist

← print words not in list

7

- ▶ sets
- ▶ dictionary clients
- ▶ indexing clients
- ▶ sparse vectors

Dictionary lookup

Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 1. DNS lookup.

```

% java LookupCSV ip.csv 0 1
adobe.com
192.150.18.60
www.princeton.edu
128.112.128.15
ebay.edu
Not found

% java LookupCSV ip.csv 1 0
128.112.128.15
www.princeton.edu
999.999.999.99
Not found

```

URL is key IP is value

IP is key URL is value

```

% more ip.csv
www.princeton.edu,128.112.128.15
www.cs.princeton.edu,128.112.136.35
www.math.princeton.edu,128.112.18.11
www.cs.harvard.edu,140.247.50.127
www.harvard.edu,128.103.60.24
www.yale.edu,130.132.51.8
www.econ.yale.edu,128.36.236.74
www.cs.yale.edu,128.36.229.30
espn.com,199.181.135.201
yahoo.com,66.94.234.13
msn.com,207.68.172.246
google.com,64.233.167.99
baidu.com,202.108.22.33
yahoo.co.jp,202.93.91.141
sina.com.cn,202.108.33.32
ebay.com,66.135.192.87
adobe.com,192.150.18.60
163.com,220.181.29.154
passport.net,65.54.179.226
tom.com,61.135.158.237
nate.com,203.226.253.11
cnn.com,64.236.16.20
daum.net,211.115.77.211
blogger.com,66.102.15.100
fastclick.com,205.180.86.4
wikipedia.org,66.230.200.100
rakuten.co.jp,202.72.51.22
...

```

Dictionary lookup

Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 2. Amino acids.

```

% java LookupCSV amino.csv 0 3
ACT
Threonine
TAG
Stop
CAT
Histidine

```

codon is key name is value

```

% more amino.csv
TTT,Phe,F,Phenylalanine
TTC,Phe,F,Phenylalanine
TTA,Leu,L,Leucine
TTG,Leu,L,Leucine
TCT,Ser,S,Serine
TCC,Ser,S,Serine
TCA,Ser,S,Serine
TCG,Ser,S,Serine
TAT,Tyr,Y,Tyrosine
TAC,Tyr,Y,Tyrosine
TAA,Stop,Stop,Stop
TAG,Stop,Stop,Stop
TGT,Cys,C,Cysteine
TGC,Cys,C,Cysteine
TGA,Stop,Stop,Stop
TGG,Trp,W,Tryptophan
CTT,Leu,L,Leucine
CTC,Leu,L,Leucine
CTA,Leu,L,Leucine
CTG,Leu,L,Leucine
CCT,Pro,P,Proline
CCC,Pro,P,Proline
CCA,Pro,P,Proline
CCG,Pro,P,Proline
CAT,His,H,Histidine
CAC,His,H,Histidine
CAA,Gln,Q,Glutamine
CAG,Gln,Q,Glutamine
CGT,Arg,R,Arginine
CGC,Arg,R,Arginine
...

```

Dictionary lookup

Command-line arguments.

- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 3. Class list.

```

% java LookupCSV classlist.csv 4 1
eberl
Ethan
nwebb
Natalie

% java LookupCSV classlist.csv 4 3
dpan
P01

```

login is key first name is value

login is key precept is value

```

% more classlist.csv
13,Berl,Ethan Michael,P01,eberl
11,Bourque,Alexander Joseph,P01,abourque
12,Cao,Phillips Minghua,P01,pcao
11,Chehoud,Christel,P01,cchehoud
10,Douglas,Malia Morioka,P01,malia
12,Haddock,Sara Lynn,P01,shaddock
12,Hantman,Nicole Samantha,P01,nhantman
11,Hesterberg,Adam Classen,P01,ahesterb
13,Hwang,Roland Lee,P01,rhwang
13,Hyde,Gregory Thomas,P01,ghyde
13,Kim,Hyunmoon,P01,hktwo
11,Kleinfeld,Ivan Maximillian,P01,ikleinfel
12,Korac,Damjan,P01,dkorac
11,MacDonald,Graham David,P01,gmacdona
10,Michal,Brian Thomas,P01,bmichal
12,Nam,Seung Hyeon,P01,seungnam
11,Nastasescu,Maria Monica,P01,mnastase
11,Pan,Di,P01,dpan
12,Partridge,Brenton Alan,P01,bpartrid
13,Rilee,Alexander,P01,arilee
13,Roopakalu,Ajay,P01,aroopaka
11,Sheng,Ben C,P01,bsheng
12,Webb,Natalie Sue,P01,nwebb
...

```

Dictionary lookup: Java implementation

```
public class LookupCSV
{
    public static void main(String[] args)
    {
        In in = new In(args[0]);
        int keyField = Integer.parseInt(args[1]);
        int valField = Integer.parseInt(args[2]);

        ST<String, String> st = new ST<String, String>();
        while (!in.isEmpty())
        {
            String line = in.readLine();
            String[] tokens = database[line].split(",");
            String key = tokens[keyField];
            String val = tokens[valField];
            st.put(key, val);
        }

        while (!StdIn.isEmpty())
        {
            String s = StdIn.readString();
            if (!st.contains(s)) StdOut.println("Not found");
            else
                StdOut.println(st.get(s));
        }
    }
}
```

process input file

build symbol table

process lookups with standard I/O

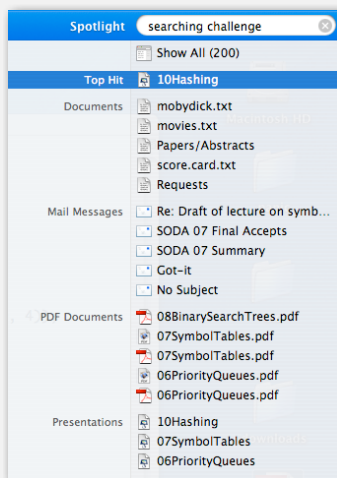
12

- sets
- dictionary clients
- indexing clients
- sparse vectors

13

File indexing

Goal. Index a PC (or the web).



14

File indexing

Goal. Given a list of files specified as command-line arguments, create an index so that you can efficiently find all files containing a given query string.

```
% ls *.txt
aesop.txt magna.txt moby.txt
sawyer.txt tale.txt

% java FileIndex *.txt
freedom
magna.txt moby.txt tale.txt

whale
moby.txt

lamb
sawyer.txt aesop.txt
```

```
% ls *.java
% java FileIndex *.java
BlackList.java Concordance.java
DeDup.java FileIndex.java ST.java
SET.java WhiteList.java

import
FileIndex.java SET.java ST.java

Comparator
null
```

Solution. Key = query string; value = set of files containing that string.

15

```

public class FileIndex
{
    public static void main(String[] args)
    {
        ST<String, SET<File>> st = new ST<String, SET<File>>();

        for (String filename : args) {
            File file = new File(filename);
            In in = new In(file);
            while (!in.isEmpty())
            {
                String word = in.readString();
                if (!st.contains(word))
                    st.put(s, new SET<File>());
                SET<File> set = st.get(key);
                set.add(file);
            }
        }

        while (!StdIn.isEmpty())
        {
            String query = StdIn.readString();
            StdOut.println(st.get(query));
        }
    }
}

```

symbol table

list of file names
from command linefor each word in file,
add file to
corresponding set

process queries

16

Goal. Index for an e-book.

Index	
Abstract data type (ADT), 127-195	stack of int (intStack), 140
abstract classes, 163	symbol table (ST), 503
classes, 129-136	text index (TI), 525
collections of items, 137-139	union-find (UF), 159
creating, 157-164	Abstract in-place merging, 351-353
defined, 128	Abstract operation, 10
duplicate items, 173-176	Access control state, 131
equivalence-relations, 159-162	Actual data, 31
FIFO queues, 165-171	Adapter class, 155-157
first-class, 177-186	Adaptive sort, 268
generic operations, 273	Address, 84-85
index items, 177	Adjacency list, 120-123
insert/remove operations, 138-139	depth-first search, 231-256
modular programming, 135	Adjacency matrix, 120-122
polynomial, 188-192	Aho, M., 464
priority queues, 375-376	Algorithm, 4-6, 27-64
pushdown stack, 138-156	abstract operations, 10, 31, 34-35
subs, 138	analysis of, 6
symbol table, 497-506	average-worst-case performance, 35, 60-62
ADT interfaces	big-O notation, 44-47
array (myArray), 274	binary search, 56-59
complex number (Complex), 181	computational complexity, 62-64
existence table (ET), 663	efficiency, 6, 30, 32
full priority queue (PQfull), 397	empirical analysis, 30-32, 58
indirect priority queue (PQind), 403	exponential-time, 219
item (myItem), 273, 498	implementation, 28-30
key (myKey), 498	logarithm function, 40-43
polynomial (poly), 189	mathematical analysis, 33-36, 58
point (Point), 114	primary parameter, 36
priority queue (PQ), 375	probabilistic, 331
queue of int (intQueue), 166	recurrences, 49-52, 57
	recursive, 198
	running time, 34-40
	search, 53-56, 498
	steps in, 22-23
	See also Randomized algorithm
	Amortization approach, 57, 627
	Arithmetic operator, 177-179, 188, 191
	Array, 12, 83
	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, 120-124
	vectors, 87
	visualizations, 295
	See also Index, array
	Array representation
	binary tree, 381
	BFS queue, 168-169
	linked lists, 110
	polynomial ADT, 191-192
	priority queue, 377-378, 403, 406
	pushdown stack, 148-150
	random queue, 170
	symbol table, 508, 511-512, 521
	Asymptotic expression, 45-46
	Average deviation, 80-81
	Average-case performance, 35, 60-61
	AVL tree, 583
	B tree, 584, 692-704
	external/internal pages, 695
	4-5-6-7-8 tree, 693-704
	Markov chain, 701
	remove, 701-703
	search/insert, 697-701
	select/insert, 701
	Balanced tree, 238, 555-598
	B tree, 584
	bottom-up, 576, 584-585
	boggle-balanced, 583
	indexed sequential access, 690-692
	performance, 575-576, 581-582, 593-598
	randomized, 559-564
	red-black, 577-585
	skip lists, 587-594
	splay, 566-571

727

17

Concordance

Goal. Preprocess a text corpus to support concordance queries: given a word, find all occurrences with their immediate contexts.

```

% java Concordance tale.txt
cities
tongues of the two *cities* that were blended in

majesty
their turnkeys and the *majesty* of the law fired
me treason against the *majesty* of the people in
of his most gracious *majesty* king george the third

princeton
no matches

```

18

Concordance

```

public class Concordance
{
    public static void main(String[] args)
    {
        In in = new In(args[0]);
        String[] words = StdIn.readAll().split("\\s+");
        ST<String, SET<Integer>> 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);
            set.put(i);
        }

        while (!StdIn.isEmpty())
        {
            String query = StdIn.readString();
            SET<Integer> set = st.get(query);
            for (int k : set)
                // print words[k-5] to words[k+5]
        }
    }
}

```

read text and
build indexprocess queries
and print
concordances

19

- ▶ sets
- ▶ dictionary clients
- ▶ indexing clients
- ▶ sparse vectors

Matrix-vector multiplication (standard implementation)

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

```

...
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;
}

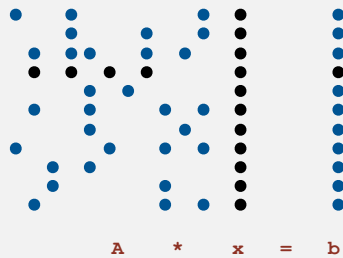
```

nested loops
(N² running time)

Sparse matrix-vector multiplication

Problem. Sparse matrix-vector multiplication.

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



Vector representations

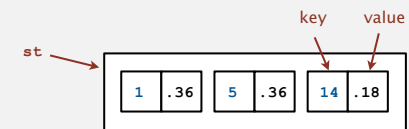
1D array (standard) representation.

- 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 : 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

24

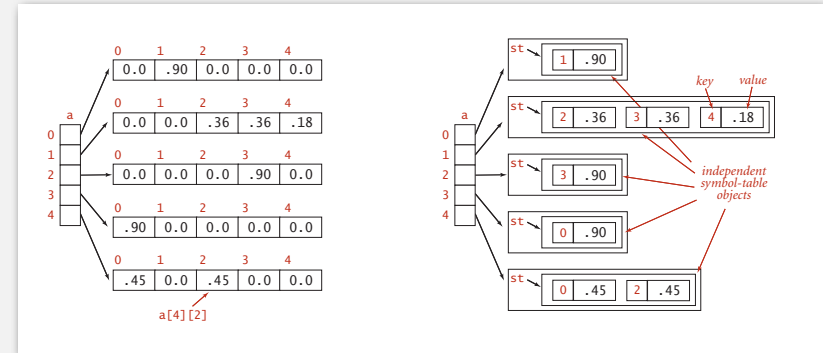
Matrix representations

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

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

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

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



25

Sparse matrix-vector multiplication

a[][]	x[]	b[]																																			
<table border="1"> <tr><td>0</td><td>.90</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>.36</td><td>.36</td><td>.18</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>.90</td><td>0</td></tr> <tr><td>.90</td><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>.47</td><td>0</td><td>.47</td><td>0</td><td>0</td></tr> </table>	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	<table border="1"> <tr><td>.05</td></tr> <tr><td>.04</td></tr> <tr><td>.36</td></tr> <tr><td>.37</td></tr> <tr><td>.19</td></tr> </table>	.05	.04	.36	.37	.19	<table border="1"> <tr><td>.036</td></tr> <tr><td>.297</td></tr> <tr><td>.333</td></tr> <tr><td>.045</td></tr> <tr><td>.1927</td></tr> </table>	.036	.297	.333	.045	.1927
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																																	
.05																																					
.04																																					
.36																																					
.37																																					
.19																																					
.036																																					
.297																																					
.333																																					
.045																																					
.1927																																					

=

```

..
SparseVector[] 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);

```

← linear running time for sparse matrix

26