3.4 HASH TABLES

- hash functions
- separate chaining
- linear probing
- context

ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>worst-case cost (after N inserts)</th>
<th>average-case cost (after N random inserts)</th>
<th>ordered iteration?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.38 lg N</td>
</tr>
<tr>
<td>red-black BST</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>1.00 lg N</td>
</tr>
</tbody>
</table>

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.

$$\text{hash("it")} = 3$$

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).
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.

Ex 1. Phone numbers.
- Bad: first three digits.
- Better: last three digits.

Ex 2. Social Security numbers.
- 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`.
Legal (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

Java library implementations

```
class Integer {
    private int value;
    public int hashCode() { return value; }
}

class Double {
    private double value;
    public int hashCode() {
        return (int) (bits ^ (bits >>> 32));
    }
}

class Boolean {
    private boolean value;
    public int hashCode() {
        return (value) ? 1231 : 1237;
    }
}
```

impliment file library implementation

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

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

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)
Implementing hash code: strings

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

```java
public final class String {
    private int hash = 0;
    private final char[] s;
    
    public int hashCode() {
        int h = hash;
        if (h != 0) return h;
        for (int i = 0; i < length(); i++)
            h = 31 * h + (31 * hash);
        return h;
    }
}
```

---

Implementing hash code: user-defined types

```java
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;
        hash = 31 * hash + who.hashCode();
        hash = 31 * hash + when.hashCode();
        hash = 31 * hash + ((Double) amount).hashCode();
        return hash;
    }
}
```

---

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()`.
- If field is an array, apply to each entry.

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.

---

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).

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

1-in-a-billion bug

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

Correct

```java
private int hash(Key key) {
    return (key.hashCode() & 0xffffffff) % M;
}
```
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 $\geq 1$ 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

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 are evenly distributed.

**Challenge.** Deal with collisions efficiently.
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.

Separate chaining ST: Java implementation

```java
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;
    }

    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]);
    }
}
```

Analysis of separate chaining

**Proposition.** Under uniform hashing assumption, prob. 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.
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 hash table demo

Hash. Map key to integer $i$ between 0 and $M-1$.
Insert. Put at table index $i$ if free; if not try $i+1, i+2, \text{etc.}$

Linear probing hash table demo
Linear probing hash table demo

**Hash.** Map key to integer $i$ between 0 and $M-1$.

**Search.** Search table index $i$; if occupied but no match, try $i+1$, $i+2$, etc.

```
search K
hash(K) = 5
```

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
st[]
P M A C S H L E R X
```

```
M = 16
K
search miss
```

```
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;
    }
}
```

Cluster. A contiguous block of items.

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

```
Clustering
```

```
``
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 # of probes in a linear probing hash table of size $M$ that contains $N = \alpha M$ keys is:

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

search hit

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

search miss / insert

Pf. A proof sketch is given. An overview of the analysis is given.

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

<table>
<thead>
<tr>
<th>implementation</th>
<th>worst-case cost (after $N$ inserts)</th>
<th>average case (after $N$ random inserts)</th>
<th>ordered iteration?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
<td>insert</td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
</tr>
<tr>
<td>binary search</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>1.38 lg N</td>
</tr>
<tr>
<td>red-black tree</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>2 lg N</td>
<td>1.00 lg N</td>
</tr>
<tr>
<td>separate chaining</td>
<td>lg N*</td>
<td>lg N*</td>
<td>lg N*</td>
<td>3-5*</td>
</tr>
<tr>
<td>linear probing</td>
<td>lg N*</td>
<td>lg N*</td>
<td>lg N*</td>
<td>3-5*</td>
</tr>
</tbody>
</table>

* under uniform hashing assumption

3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- context
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.

```java
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/12type/index.html

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.

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.

<table>
<thead>
<tr>
<th>key</th>
<th>hashCode()</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Aa&quot;</td>
<td>2112</td>
</tr>
<tr>
<td>&quot;BB&quot;</td>
<td>2112</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>key</th>
<th>hashCode()</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;AaAaAaAa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaAaAaBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaAaBBaa&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaAaBBBB&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBBBBaA&quot;</td>
<td>-540425984</td>
</tr>
<tr>
<td>&quot;AaBBBBBB&quot;</td>
<td>-540425984</td>
</tr>
</tbody>
</table>

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.

Q. How to delete?
Q. How to resize?

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.

Hash tables vs. balanced search trees

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.

3.5 Symbol Table Applications
- sets
- dictionary clients
- indexing clients
- sparse vectors
3.5 SYMBOL TABLE APPLICATIONS

- sets
- dictionary clients
- indexing clients
- sparse vectors

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?

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
```

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.

```
<table>
<thead>
<tr>
<th>application</th>
<th>purpose</th>
<th>key</th>
<th>in list</th>
</tr>
</thead>
<tbody>
<tr>
<td>spell checker</td>
<td>identify misspelled words</td>
<td>word</td>
<td>dictionary words</td>
</tr>
<tr>
<td>browser</td>
<td>mark visited pages</td>
<td>URL</td>
<td>visited pages</td>
</tr>
<tr>
<td>parental controls</td>
<td>block sites</td>
<td>URL</td>
<td>bad sites</td>
</tr>
<tr>
<td>chess</td>
<td>detect draw</td>
<td>board</td>
<td>positions</td>
</tr>
<tr>
<td>spam filter</td>
<td>eliminate spam</td>
<td>IP address</td>
<td>spam addresses</td>
</tr>
<tr>
<td>credit cards</td>
<td>check for stolen cards</td>
<td>number</td>
<td>stolen cards</td>
</tr>
</tbody>
</table>
```
### 3.5 Symbol Table Applications

- **sets**
- **dictionary clients**
- **indexing clients**
- **sparse vectors**

#### Exception filter: Java implementation

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

```java
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);
        }
    }
}
```

#### BlackList

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

```java
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);
        }
    }
}
```

#### Dictionary lookup

- **Command-line arguments.**
  - A comma-separated value (CSV) file.
  - Key field.
  - Value field.

**Ex 1. DNS lookup.**

```bash
% java LookupCSV ip.csv 0 1
adobe.com 192.150.18.60
www.princeton.edu 128.112.128.15
ebay.com 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**

**URL is key, IP is value**

**IP is key, URL is value**

**IP is key, URL is value**
Command-line arguments.
- A comma-separated value (CSV) file.
- Key field.
- Value field.

Ex 2. Amino acids.

<table>
<thead>
<tr>
<th>codon</th>
<th>name</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACT</td>
<td>Threonine</td>
<td></td>
</tr>
<tr>
<td>TAG</td>
<td>Histidine</td>
<td></td>
</tr>
<tr>
<td>CAT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ex 3. Class list.

<table>
<thead>
<tr>
<th>login</th>
<th>first name</th>
<th>section</th>
</tr>
</thead>
<tbody>
<tr>
<td>eberl</td>
<td>Ethan</td>
<td></td>
</tr>
<tr>
<td>nwebb</td>
<td>Natalie</td>
<td></td>
</tr>
</tbody>
</table>

Dictionary lookup: Java implementation

```java
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 = line.split(",");
            String key = tokens[keyField];
            String val = tokens[valField];
            st.put(key, val);
        }

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

3.5 SYMBOL TABLE APPLICATIONS
Goal. Index a PC (or the web).

File indexing

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

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

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

File indexing

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

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

Book index

Goal. Index for an e-book.
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

---

**public class Concordance**

```java
public class Concordance {
    public static void main(String[] args) {
        In in = new In(args[0]);
        String[] words = StdIn.readStrings();
        ST<String, SET<Integer>> st = new ST<String, SET<Integer>>();
        for (int i = 0; i < words.length; i++)
            if (st.contains(words[i]))
                st.put(words[i], new SET<Integer>());
        while (!StdIn.isEmpty())
            String query = StdIn.readString();
            SET<Integer> set = st.get(query);
            for (int k : set)
                // print words[k-4] to words[k+4]
    }
}
```

---

**Matrix-vector multiplication (standard implementation)**

```java
double[][] a = new double[N][N];
double[] x = new double[N];
double[] b = new double[N];
...  
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;
    }
```
Sparse matrix-vector multiplication

**Problem.** Sparse matrix-vector multiplication.

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

\[ A \times x \times b \]

Sparse matrix-vector multiplication

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 | .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

```java
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;
    }
}
```

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 \( N \)).
Sparse matrix-vector multiplication

```
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