3.4 Hash Tables

- hash functions
- separate chaining
- linear probing
- applications

ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
</tr>
<tr>
<td>(linked list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>lg N</td>
<td>N</td>
<td>N</td>
<td>lg N</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></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>
</tbody>
</table>

Q. Can we do better?
A. Yes, but with different access to the data.

Optimize judiciously

- More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason—including blind stupidity. — William A. Wulf
- We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. — Donald E. Knuth
- We follow two rules in the matter of optimization:
  Rule 1: Don’t do it.
  Rule 2 (for experts only). Don’t do it yet - that is, not until you have a perfectly clear and unoptimized solution. — M. A. Jackson

Reference: Effective Java by Joshua Bloch

Hashing: basic plan

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

Hash function. Method for computing 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.

<table>
<thead>
<tr>
<th>key</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;it&quot;</td>
<td>3</td>
</tr>
<tr>
<td>&quot;times&quot;</td>
<td>3</td>
</tr>
</tbody>
</table>

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`.

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

```java
class Integer {
    private final int value;
    ...}

class Double {
    private final double value;
    ...
    public int hashCode() {
        long bits = doubleToLongBits(value);
        return (int) (bits ^ (bits >>> 32));
    }
}
```

```java
class Boolean {
    private final boolean value;
    ...
    public int hashCode() {
        if (value) return 1231;
        else return 1237;
    }
}
```

```java
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;
    }
}
```

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

Implementing hash code: strings

- Horner’s method to hash string of length L: multiplies/adds.
- Equivalent to $h = 31^{L-1} \cdot s_0 + \ldots + 31^2 \cdot s_3 + 31^1 \cdot s_2 + 31^0 \cdot s_1$.

```java
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… …cac vtcbc vuccc vv… kkk
```

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

```
http://www.cs.princeton.edu/introcs/12type/index.html
```

```
War story: String hashing in 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;
}
```

```
public final class Transaction {   private final long who;   private final Date when;   private final String where;   ...
```

```
nonzero constant
```

```
for primitive types, use hashCode() of wrapper type
```

```
for reference types, use hashCode()
```

```
typically a small prime
```

```
public int hashCode() {
    int hash = 17;
    hash = 31*hash + ((Long) val).hashCode();
    hash = 31*hash + when.hashCode();
    hash = 31*hash + where.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 an array, apply to each element.
• If field is a reference type, use `hashCode()`.

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 $-2^{31}$ 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

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 $\sim \sqrt{M / 2}$ tosses.

Coupon collector. Expect every bin has $\approx 1$ 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 $\Rightarrow$ 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 $i^{th}$ chain.

```java
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() {
        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);
    }
}
```
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.

ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Ordered Iteration?</th>
<th>Operations on Keys</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>lg N/2</td>
<td>N/2</td>
</tr>
<tr>
<td>BST</td>
<td>Y</td>
<td>N</td>
<td>1.38 lg 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>1.00 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>
</tbody>
</table>

*: under uniform hashing assumption

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.
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 ST implementation

```java
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

**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:

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


Parameters.
- \( M \) too large \( \Rightarrow \) too many empty array entries.
- \( M \) too small \( \Rightarrow \) search time blows up.
- Typical choice: \( \alpha = N/M \sim \frac{1}{2} \).

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>-5404255984</td>
</tr>
<tr>
<td>&quot;AaAaAaBB&quot;</td>
<td>-5404255984</td>
</tr>
<tr>
<td>&quot;AaAaBBAa&quot;</td>
<td>-5404255984</td>
</tr>
<tr>
<td>&quot;AaAaBBBB&quot;</td>
<td>-5404255984</td>
</tr>
<tr>
<td>&quot;AaBBAaAa&quot;</td>
<td>-5404255984</td>
</tr>
<tr>
<td>&quot;AaBBAaBB&quot;</td>
<td>-5404255984</td>
</tr>
<tr>
<td>&quot;AaBBBBBa&quot;</td>
<td>-5404255984</td>
</tr>
<tr>
<td>&quot;AaBBBBBB&quot;</td>
<td>-5404255984</td>
</tr>
</tbody>
</table>

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.
Memory usage for ST implementations

Separate chaining
- N nodes (each 24 bytes overhead plus three 8-byte references)
- M nodes (each 24 bytes overhead plus one 8-byte reference)
- Total: $\approx 48N + 32M = \approx 54.4N$ bytes for recommended $M = N/5$

Linear probing
- 2N references if 1/2 full
- 8N references if 1/8 full
- Total: between $\approx 32N$ and $\approx 128N$ bytes

Binary search trees
- N nodes (each 24 bytes overhead plus four 8-byte references)
- Total: $\approx 56N$ bytes

Bottom line: Memory usage not decisive in reference implementations

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).
- Not easy to implement `equals()` and `hashCode()` correctly.

Balanced search trees.
- Stronger performance guarantee.
- Support for ordered ST operations.
- Faster for complicated keys (compare vs. arithmetic ops on whole key)
- Not difficult to implement `compareTo()` correctly.

Java system includes both.
- Red-black trees: `java.util.TreeMap`, `java.util TreeSet`
- Hashing: `java.util.HashMap`, `java.util.IdentityHashMap`
3.5 Symbol Table Applications

- sets
- dictionary clients
- indexing clients
- sparse vectors
- geometric applications

Set API

**Mathematical set.** A collection of distinct keys.

```java
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.

```bash
% 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
% 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>

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.

```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);
        }
    }
}
```

```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);
        }
    }
}
```

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.

```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);
        }
    }
}
```

- 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
192.150.18.60
www.princeton.edu

% java LookupCSV ip.csv 1 0
128.112.128.15
www.princeton.edu

% java LookupCSV classlist.csv 4 1
Berl,Ethan Michael,P01,eberl
Bourque,Alexander Joseph,P01,abourque
Boumas,P01,mboumas
Cao,Phillips P,P01,pcao
Chehoud,Christel,P01,chezhou
Douglas,Malia Moraic,P01,madou
Dodd,Dara Lynn,P01,shdodd
Hantsen,Nicole Samantha,P01,hantse
Hastenberg,Adam Classes,P01,ashesten
Haeng,Roland Lee,P01,haeng
Hyde,Jeremy Thomas,P01,hyde
Kim,Hyunmoon,P01,hkim
Kleinfield,Ivan Maximilian,P01,ikleinfe
Korser,Derjen,P01,dkorser
MacDonald,Graham David,P01,gmaconna
Michal,Brian Thomas,P01,bmichal
Nam,Saung Byoung,P01,nambyou
Nastassova,Maria Monica,P01,mnastas
Pan,Di,P01,dpang
Partridge,Branton Alan,P01,bpatrid
Riley,Alexander P01,ariley
Rongpaisal,Apjay,P01,arongpa
Sheng,Ben C,P01,bsheng
Webb,Natalie Sue,P01,nwebb

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

% java LookupCSV classlist.csv 4 3
Dpan,P01
Nwebb,P01

% java LookupCSV ip.csv
192.150.18.60
www.princeton.edu
%

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>();
      process input file
      build symbol table
      process lookups with standard I/O

      while (!in.isEmpty()) {
         String line = in.readLine();
         String[] tokens = database[i].split(",");
         String key = tokens[keyField];
         String val = tokens[valField];
         st.put(key, val);
         if (!st.contains(s)) StdOut.println("Not found");
         else                 StdOut.println(st.get(s));
      }
   }
}

Goal. Index a PC (or the web).

File indexing

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

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

```java
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));
        }
    }
}
```
Keyword-in-context search

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

```
% java KWIC core.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 KWIC
{
  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]
    }
  }
}
```

Matrix-vector multiplication (standard implementation)

```
... 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;
... }
```
Sparse matrix-vector multiplication

Problem. Sparse matrix-vector multiplication.

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

Sparse matrix-vector multiplication

1D array (standard) representation.
- Constant time access to elements.
- Space proportional to N.

Symbol table representation.
- Key = index, value = entry.
- Efficient iterator.
- Space proportional to number of nonzeros.

Vector 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 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;  }
}
Sparse matrix-vector multiplication

\[ a[i] \times x[i] = b[i] \]

\[
\begin{bmatrix}
0 & .90 & 0 & 0 & 0 \\
0 & 0 & .36 & .36 & .18 \\
0 & 0 & 0 & .90 & 0 \\
.90 & 0 & 0 & 0 & 0 \\
.47 & 0 & .47 & 0 & 0
\end{bmatrix}
\times
\begin{bmatrix}
.05 \\
.04 \\
.36 \\
.37 \\
.19
\end{bmatrix}
= 
\begin{bmatrix}
.036 \\
.297 \\
.331 \\
.045 \\
.927
\end{bmatrix}
\]

2d tree

Recursively partition plane into two halfplanes.

2d tree implementation

Data structure. BST, but alternate using \( x \)- and \( y \)-coordinates as key.
- Search gives rectangle containing point.
- Insert further subdivides the plane.
2d tree: 2d orthogonal range search

Range search. Find all points in a query axis-aligned rectangle.
- Check if point in node lies in given rectangle.
- Recursively search left/top subdivision (if any could fall in rectangle).
- Recursively search right/bottom subdivision (if any could fall in rectangle).

Typical case. $R + \log N$.
Worst case (assuming tree is balanced). $R + \sqrt{N}$.

Kd tree

Kd tree. Recursively partition $k$-dimensional space into 2 halfspaces.

Implementation. BST, but cycle through dimensions ala 2d trees.

Efficient, simple data structure for processing $k$-dimensional data.
- Widely used.
- Adapts well to high-dimensional and clustered data.
- Discovered by an undergrad (Jon Bentley) in an algorithms class!

Search for intersections

Problem. Find all intersecting pairs among $N$ geometric objects.
Applications. CAD, games, movies, virtual reality, ....

Simple version. 2d, all objects are horizontal or vertical line segments.

Brute force. Test all $\Theta(N^2)$ pairs of line segments for intersection.
Orthogonal line segment intersection search: sweep-line algorithm

Sweep vertical line from left to right.
- \( x \)-coordinates define events.
- \( h \)-segment (left endpoint): insert \( y \)-coordinate into ST.
- \( h \)-segment (right endpoint): remove \( y \)-coordinate from ST.

Proposition. The sweep-line algorithm takes time proportional to \( N \log N + R \) to find all \( R \) intersections among \( N \) orthogonal segments.
- Put \( x \)-coordinates on a PQ (or sort). \( N \log N \)
- Insert \( y \)-coordinates into ST. \( N \log N \)
- Delete \( y \)-coordinates from ST. \( N \log N \)
- Range searches. \( N \log N + R \)

Efficiency relies on judicious use of data structures.

Remark. Sweep-line solution extends to 3d and more general shapes.
Searching challenge 1A

Problem. Maintain symbol table of song names for an iPod.
Assumption A. Hundreds of songs.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn’t matter much, all fast enough.

Searching challenge 1B

Problem. Maintain symbol table of song names for an iPod.
Assumption B. Thousands of songs.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn’t matter much, all fast enough.

Searching challenge 2A

Problem. IP lookups in a web monitoring device.
Assumption A. Billions of lookups, millions of distinct addresses.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn’t matter much, all fast enough.
Searching challenge 2B

Problem. IP lookups in a web monitoring device.
Assumption B. Billions of lookups, thousands of distinct addresses.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) Need better method, all too slow.
4) Doesn’t matter much, all fast enough.

Searching challenge 3

Problem. Frequency counts in "Tale of Two Cities."
Assumptions. Book has 135,000+ words; about 10,000 distinct words.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) BSTs.
4) Hashing.

Searching challenge 4

Problem. Spell checking for a book.
Assumptions. Dictionary has 25,000 words; book has 100,000+ words.

Which searching method to use?
1) Sequential search in a linked list.
2) Binary search in an ordered array.
3) BST.
4) Hashing.

Searching challenge 5

Problem. Index for a PC or the web.
Assumptions. 1 billion++ words to index.

Which searching method to use?
• Hashing
• LLRB trees
• Doesn’t matter much.
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.