Hash

- 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

hash functions

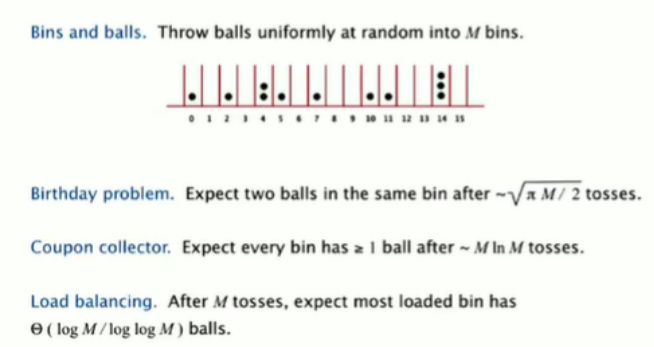
Goal

- Scramble the keys uniformly to produce a table index
  - efficiently computable
  - each table index equally likely for each other

Java's hash code

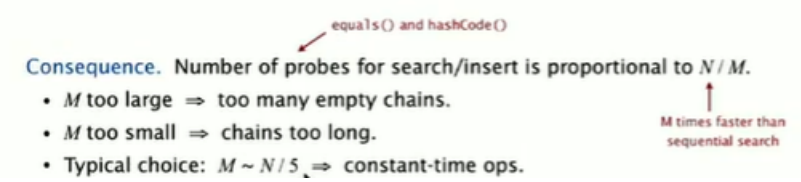
```
if x.equals(y), then x.hashCode() == y.hashCode()
```

Uniform hashing assumption



separate chaining

Collision



Implementation

```
public class SeparateChainingHashST<Key, Value> {
    private static final int INIT_CAPACITY = 4;

    private int n;                                // number of key-value pairs
```

```
    private int m;                                // hash table size
    private SequentialSearchST<Key, Value>[] st;  // array of linked-list
symbol tables

    public SeparateChainingHashST() {
        this(INIT_CAPACITY);
    }

    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 void resize(int chains) {
        SeparateChainingHashST<Key, Value> temp = new
SeparateChainingHashST<Key, Value>(chains);
        for (int i = 0; i < m; i++) {
            for (Key key : st[i].keys()) {
                temp.put(key, st[i].get(key));
            }
        }
        this.m  = temp.m;
        this.n  = temp.n;
        this.st = temp.st;
    }

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

    private int hash(Key key) {
        int h = key.hashCode();
        h ^= (h >>> 20) ^ (h >>> 12) ^ (h >>> 7) ^ (h >>> 4);
        return h & (m-1);
    }

    public int size() {
        return n;
    }

    public boolean isEmpty() {
        return size() == 0;
    }

    public boolean contains(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to
contains() is null");
        return get(key) != null;
    }

```

```
    public Value get(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to get() is null");
        int i = hash(key);
        return st[i].get(key);
    }

    public void put(Key key, Value val) {
        if (key == null) throw new IllegalArgumentException("first argument to put() is null");
        if (val == null) {
            delete(key);
            return;
        }

        if (n >= 10*m) resize(2*m);

        int i = hash(key);
        if (!st[i].contains(key)) n++;
        st[i].put(key, val);
    }

    public void delete(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to delete() is null");

        int i = hash(key);
        if (st[i].contains(key)) n--;
        st[i].delete(key);

        if (m > INIT_CAPACITY && n <= 2*m) resize(m/2);
    }

    public Iterable<Key> keys() {
        Queue<Key> queue = new Queue<Key>();
        for (int i = 0; i < m; i++) {
            for (Key key : st[i].keys())
                queue.enqueue(key);
        }
        return queue;
    }

    public static void main(String[] args) {
        SeparateChainingHashST<String, Integer> st = new SeparateChainingHashST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);

        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }D
```

```
}
```

linear probing

Open addressing

> when a new key collides, find next empty slot, and put it there

> Array size M must be greater than number of key-value pairs N

Complexity

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

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

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	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()

Parameters

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N / M \sim ½$. ← # probes for search hit is about 3/2; # probes for search miss is about 5/2

Implementation

```
public class LinearProbingHashST<Key, Value> {

    private static final int INIT_CAPACITY = 4;

    private int n;           // number of key-value pairs in the symbol table
    private int m;           // size of linear probing table
    private Key[] keys;      // the keys
```

```
    private Value[] vals;    // the values

    public LinearProbingHashST() {
        this(INIT_CAPACITY);
    }

    public LinearProbingHashST(int capacity) {
        m = capacity;
        n = 0;
        keys = (Key[])   new Object[m];
        vals = (Value[]) new Object[m];
    }

    public int size() {
        return n;
    }

    public boolean isEmpty() {
        return size() == 0;
    }

    public boolean contains(Key key) {
        if (key == null) throw new IllegalArgumentException("argument to
contains() is null");
        return get(key) != null;
    }

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

    private int hash(Key key) {
        int h = key.hashCode();
        h ^= (h >>> 20) ^ (h >>> 12) ^ (h >>> 7) ^ (h >>> 4);
        return h & (m-1);
    }

    private void resize(int capacity) {
        LinearProbingHashST<Key, Value> temp = new LinearProbingHashST<Key,
Value>(capacity);
        for (int i = 0; i < m; i++) {
            if (keys[i] != null) {
                temp.put(keys[i], vals[i]);
            }
        }
        keys = temp.keys;
        vals = temp.vals;
        m    = temp.m;
    }

    public void put(Key key, Value val) {
```

```
57          if (key == null) throw new IllegalArgumentException("first argument to
    put() is null");
58
59          if (val == null) {
60              delete(key);
61              return;
62          }
63
64          if (n >= m/2) resize(2*m);
65
66          int i;
67          for (i = hash(key); keys[i] != null; i = (i + 1) % m) {
68              if (keys[i].equals(key)) {
69                  vals[i] = val;
70                  return;
71              }
72          }
73          keys[i] = key;
74          vals[i] = val;
75          n++;
76      }
77
78      public Value get(Key key) {
79          if (key == null) throw new IllegalArgumentException("argument to get()
    is null");
80          for (int i = hash(key); keys[i] != null; i = (i + 1) % m)
81              if (keys[i].equals(key))
82                  return vals[i];
83          return null;
84      }
85
86      public void delete(Key key) {
87          if (key == null) throw new IllegalArgumentException("argument to
    delete() is null");
88          if (!contains(key)) return;
89
90          int i = hash(key);
91          while (!key.equals(keys[i])) {
92              i = (i + 1) % m;
93          }
94
95          keys[i] = null;
96          vals[i] = null;
97
98          i = (i + 1) % m;
99          while (keys[i] != null) {
100             Key   keyToRehash = keys[i];
101             Value valToRehash = vals[i];
102             keys[i] = null;
103             vals[i] = null;
104             n--;
105             put(keyToRehash, valToRehash);
```

```
            i = (i + 1) % m;
        }

        n--;

        if (n > 0 && n <= m/8) resize(m/2);

        assert check();
    }

    public Iterable<Key> keys() {
        Queue<Key> queue = new Queue<Key>();
        for (int i = 0; i < m; i++)
            if (keys[i] != null) queue.enqueue(keys[i]);
        return queue;
    }

    private boolean check() {

        if (m < 2*n) {
            System.err.println("Hash table size m = " + m + "; array size n = " + n);
            return false;
        }

        for (int i = 0; i < m; i++) {
            if (keys[i] == null) continue;
            else if (get(keys[i]) != vals[i]) {
                System.err.println("get[" + keys[i] + "] = " + get(keys[i]) + "; vals[i] = " + vals[i]);
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        LinearProbingHashST<String, Integer> st = new LinearProbingHashST<String, Integer>();
        for (int i = 0; !StdIn.isEmpty(); i++) {
            String key = StdIn.readString();
            st.put(key, i);
        }

        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }
}
```

Comparision

- separate chaining
    - easier to implement delete
    - performance degrades gracefully
    - clustering less sensitive to poorly-designed has funtion
- linear probing
    - less wasted space
    - better cache performance

context

One-way hash function

> hard to find a key that will hash to a desired value (or two keys that hash to the same value)

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

Hashing: variations on the theme

Two-probe hashing

- hash two positions, insert key in shorter of the two chains
- reduces expected length of the longest chain to $log\ logN$

Double hashing

- 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

Cukoo hashing

- hash keys 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 Table vs. BST

- hash table
    - simpler to code
    - no effective alternative for unordered keys
    - faster for simple keys
    - better system support in java for strings
- BST
    - stronger performance guarantee

- support for ordered ST operations
- easier to implement $compareTo()$ correctly than $equals()$ and $hashCode()$

Symbol Table Applications

Set

> a collection of distinct keys

API

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*

Filter

```java
public class BlockFilter {

    // Do not instantiate.
    private BlockFilter() { }

    public static void main(String[] args) {
        SET<String> set = new SET<String>();

        // read in strings and add to set
        In in = new In(args[0]);
        while (!in.isEmpty()) {
            String word = in.readString();
            set.add(word);
        }

        // read in string from standard input, printing out all exceptions
        while (!StdIn.isEmpty()) {
            String word = StdIn.readString();
            if (!set.contains(word))
                StdOut.println(word);
        }
    }
}
```

Dictionary

LookupCSV

```
public class LookupCSV {

    // Do not instantiate.
    private LookupCSV() { }

    public static void main(String[] args) {
        int keyField = Integer.parseInt(args[1]);
        int valField = Integer.parseInt(args[2]);

        // symbol table
        ST<String, String> st = new ST<String, String>();

        // read in the data from csv file
        In in = new In(args[0]);
        while (in.hasNextLine()) {
            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.contains(s)) StdOut.println(st.get(s));
            else                StdOut.println("Not found");
        }
    }
}
```

Indexing

File index

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

```
import java.io.File;

public class FileIndex {

    // Do not instantiate.
    private FileIndex() { }

    public static void main(String[] args) {

        // key = word, value = set of files containing that word
        ST<String, SET<File>> st = new ST<String, SET<File>>();
```

```
12
13          // create inverted index of all files
14          StdOut.println("Indexing files");
15          for (String filename : args) {
16              StdOut.println("  " + filename);
17              File file = new File(filename);
18              In in = new In(file);
19              while (!in.isEmpty()) {
20                  String word = in.readString();
21                  if (!st.contains(word)) st.put(word, new SET<File>());
22                  SET<File> set = st.get(word);
23                  set.add(file);
24              }
25          }
26
27          while (!StdIn.isEmpty()) {
28              String query = StdIn.readString();
29              if (st.contains(query)) {
30                  SET<File> set = st.get(query);
31                  for (File file : set) {
32                      StdOut.println("  " + file.getName());
33                  }
34              }
35          }
36      }
37  }
```

Concordance

```
1   public class Concordance {
2
3       public static void main(String[] args) {
4           int CONTEXT = 5;
5
6           In in = new In(args[0]);
7           String[] words = in.readAllStrings();
8           ST<String, SET<Integer>> st = new ST<String, SET<Integer>>();
9
10          // build up concordance
11          for (int i = 0; i < words.length; i++) {
12              String s = words[i];
13              if (!st.contains(s)) {
14                  st.put(s, new SET<Integer>());
15              }
16              SET<Integer> set = st.get(s);
17              set.add(i);
18          }
19          StdOut.println("Finished building concordance");
20
21          // process queries
22          while (!StdIn.isEmpty()) {
```

```
            String query = StdIn.readString();
            SET<Integer> set = st.get(query);
            if (set == null) set = new SET<Integer>();
            for (int k : set) {
                for (int i = Math.max(0, k - CONTEXT + 1); i < k; i++)
                    StdOut.print(words[i] + " ");
                StdOut.print("*" + words[k] + "* ");
                for (int i = k + 1; i < Math.min(k + CONTEXT, words.length);
i++)
                    StdOut.print(words[i] + " ");
                StdOut.println();
            }
            StdOut.println();
        }

    }
}
```

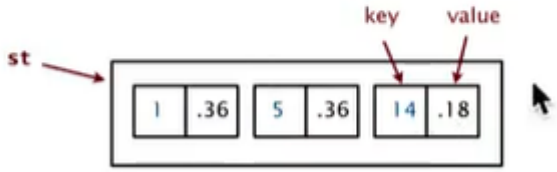
Sparse Vectors

Sparse matrix-vector multiplication

Sparse Vectors

symbol table representation

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



Implementation

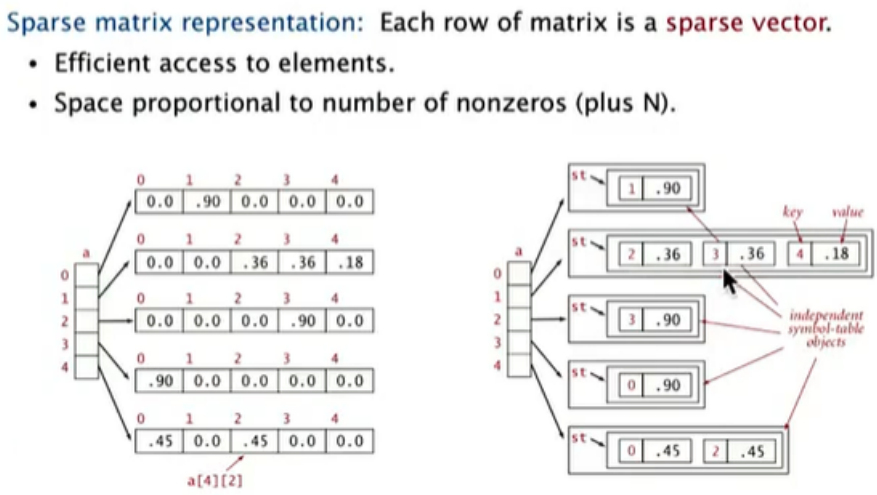
```
public class SparseVector {
    private int d;                  // dimension
    private ST<Integer, Double> st;  // the vector, represented by index-value
pairs

    public SparseVector(int d) {
        this.d  = d;
        this.st = new ST<Integer, Double>();
    }

    public void put(int i, double value) {
```

```
        if (i < 0 || i >= d) throw new IllegalArgumentException("Illegal
index");
        if (value == 0.0) st.delete(i);
        else              st.put(i, value);
    }

    public double get(int i) {
        if (i < 0 || i >= d) throw new IllegalArgumentException("Illegal
index");
        if (st.contains(i)) return st.get(i);
        else                return 0.0;
    }

    public int nnz() {
        return st.size();
    }

    @Deprecated
    public int size() {
        return d;
    }

    public int dimension() {
        return d;
    }

    public double dot(SparseVector that) {
        if (this.d != that.d) throw new IllegalArgumentException("Vector
lengths disagree");
        double sum = 0.0;

        if (this.st.size() <= that.st.size()) {
            for (int i : this.st.keys())
                if (that.st.contains(i)) sum += this.get(i) * that.get(i);
        }
        else  {
            for (int i : that.st.keys())
                if (this.st.contains(i)) sum += this.get(i) * that.get(i);
        }
        return sum;
    }

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

    public double magnitude() {
        return Math.sqrt(this.dot(this));
    }
```

```
60
61      @Deprecated
62      public double norm() {
63          return Math.sqrt(this.dot(this));
64      }
65
66      public SparseVector scale(double alpha) {
67          SparseVector c = new SparseVector(d);
68          for (int i : this.st.keys()) c.put(i, alpha * this.get(i));
69          return c;
70      }
71
72      public SparseVector plus(SparseVector that) {
73          if (this.d != that.d) throw new IllegalArgumentException("Vector
    lengths disagree");
74          SparseVector c = new SparseVector(d);
75          for (int i : this.st.keys()) c.put(i, this.get(i));                //
    c = this
76          for (int i : that.st.keys()) c.put(i, that.get(i) + c.get(i));     //
    c = c + that
77          return c;
78      }
79
80      public String toString() {
81          StringBuilder s = new StringBuilder();
82          for (int i : st.keys()) {
83              s.append("(" + i + ", " + st.get(i) + ") ");
84          }
85          return s.toString();
86      }
87
88      public static void main(String[] args) {
89          SparseVector a = new SparseVector(10);
90          SparseVector b = new SparseVector(10);
91          a.put(3, 0.50);
92          a.put(9, 0.75);
93          a.put(6, 0.11);
94          a.put(6, 0.00);
95          b.put(3, 0.60);
96          b.put(4, 0.90);
97          StdOut.println("a = " + a);
98          StdOut.println("b = " + b);
99          StdOut.println("a dot b = " + a.dot(b));
100         StdOut.println("a + b   = " + a.plus(b));
101     }
102
103 }
```

Sparse Matrix

symbol table representation



Implementation

```
1
2  public class SparseMatrix {
3      private int n;                  // n-by-n matrix
4      private SparseVector[] rows;    // the rows, each row is a sparse vector
5
6      // initialize an n-by-n matrix of all 0s
7      public SparseMatrix(int n) {
8          this.n = n;
9          rows = new SparseVector[n];
10         for (int i = 0; i < n; i++)
11             rows[i] = new SparseVector(n);
12     }
13
14     public void put(int i, int j, double value) {
15         if (i < 0 || i >= n) throw new IllegalArgumentException("Illegal
   index");
16         if (j < 0 || j >= n) throw new IllegalArgumentException("Illegal
   index");
17         rows[i].put(j, value);
18     }
19
20     public double get(int i, int j) {
21         if (i < 0 || i >= n) throw new IllegalArgumentException("Illegal
   index");
22         if (j < 0 || j >= n) throw new IllegalArgumentException("Illegal
   index");
23         return rows[i].get(j);
24     }
25
26     public int nnz() {
27         int sum = 0;
```

```
        for (int i = 0; i < n; i++)
            sum += rows[i].nnz();
        return sum;
    }

    public SparseVector times(SparseVector x) {
        if (n != x.size()) throw new IllegalArgumentException("Dimensions
disagree");
        SparseVector b = new SparseVector(n);
        for (int i = 0; i < n; i++)
            b.put(i, rows[i].dot(x));
        return b;
    }

    public SparseMatrix plus(SparseMatrix that) {
        if (this.n != that.n) throw new RuntimeException("Dimensions
disagree");
        SparseMatrix result = new SparseMatrix(n);
        for (int i = 0; i < n; i++)
            result.rows[i] = this.rows[i].plus(that.rows[i]);
        return result;
    }

    public String toString() {
        String s = "n = " + n + ", nonzeros = " + nnz() + "\n";
        for (int i = 0; i < n; i++) {
            s += i + ": " + rows[i] + "\n";
        }
        return s;
    }

    public static void main(String[] args) {
        SparseMatrix A = new SparseMatrix(5);
        SparseVector x = new SparseVector(5);
        A.put(0, 0, 1.0);
        A.put(1, 1, 1.0);
        A.put(2, 2, 1.0);
        A.put(3, 3, 1.0);
        A.put(4, 4, 1.0);
        A.put(2, 4, 0.3);
        x.put(0, 0.75);
        x.put(2, 0.11);
        StdOut.println("x     : " + x);
        StdOut.println("A     : " + A);
        StdOut.println("Ax    : " + A.times(x));
        StdOut.println("A + A : " + A.plus(A));
    }
}
```