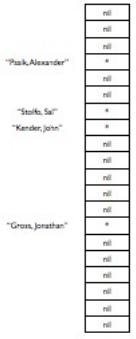
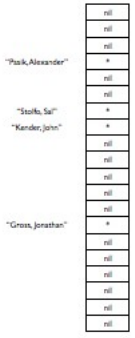
Hash Tables

Using the Data as Indices

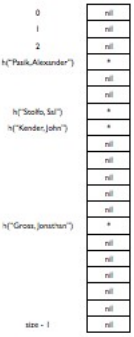


Using the Data as Indices



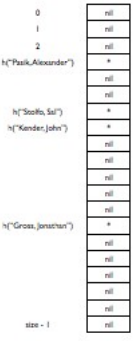
Problems:
size of domain >> universe
size of domain >> number of elements
implementing a unique index per string

Using the Data as Indices



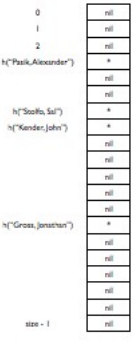
Solution:
design a function, h, called a hash code
range of h is 0..size - 1, size being reasonable
h(string) should be different for each string

Using the Data as Indices



Solution:
design a function, h, called a hash code
range of h is 0..size - 1, size being reasonable
h(string) should be different for each string
h(string) should minimize collisions
collision handling mechanism needed

Using the Data as Indices



Solution:
design a function, h, called a hash code
range of h is 0..size - 1, size being reasonable
h(string) should minimize collisions
achieved with size being prime
h(string) = numeric value of string % size
collision handling mechanism needed
make entries lists of elements

Time Complexities

	Find	Add	Memory
Unsorted Array	O(n)	O(1)	Static
Sorted Array	O(log n)	O(n)	Static
Unsorted List	O(n)	O(1)	Dynamic
Sorted List	O(n)	O(n)	Dynamic
Ordered Binary Tree	O(log n)	O(log n)	Dynamic

Time Complexities

	Find	Add	Memory
Unsorted Array	O(n)	O(1)	Static
Sorted Array	O(log n)	O(n)	Static
Unsorted List	O(n)	O(1)	Dynamic
Sorted List	O(n)	O(n)	Dynamic
Ordered Binary Tree	O(log n)	O(log n)	Dynamic
Hash Table	O(1)	O(1)	Mixed

```
public class HashData implements Comparable {
    private String str;
    private int count;
    private int mod;

    public HashData (String s, int m) {
        this.str = s; this.count = 1; this.mod = m;
    }
.
.
.
}
```

```
public class HashData implements Comparable {
.
.
.
    public boolean equals (Object d) {
        return d != null &&
               d.getClass().equals(this.getClass()) &&
               this.str.equals(((HashData) d).str);
    }

    public int compareTo (Object d) {
        return this.str.compareTo(((HashData) d).str);
    }
.
.
.
}
```

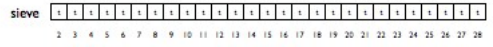
```
public class HashData implements Comparable {
.
.
.
    public String toString () { return this.str; }

    public int hashCode () {
        int x = 0;
        for (int i = 0; i < this.str.length(); i++)
            x = (((int) this.str.charAt(i)) + 256 * x) %
                this.mod;
        return x;
    }

    public void incr () { this.count++; }
}
```

Eratosthenes' Sieve

- is n prime? check by dividing n by each prime between 2 and $\sqrt{n}$
- list of primes between 2 and x can be calculated in linear time using Eratosthenes' sieve



is 2 prime?

Eratosthenes' Sieve

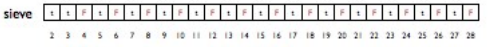
- is n prime? check by dividing n by each prime between 2 and $\sqrt{n}$
- list of primes between 2 and x can be calculated in linear time using Eratosthenes' sieve

is 2 prime?

yes -- so cross out multiples of 2

Eratosthenes' Sieve

- is n prime? check by dividing n by each prime between 2 and $\sqrt{n}$
- list of primes between 2 and x can be calculated in linear time using Eratosthenes' sieve



Eratosthenes' Sieve

- is n prime? check by dividing n by each prime between 2 and $\sqrt{n}$
- list of primes between 2 and x can be calculated in linear time using Eratosthenes' sieve

is 3 prime?

Eratosthenes' Sieve

- is n prime? check by dividing n by each prime between 2 and $\sqrt{n}$
- list of primes between 2 and x can be calculated in linear time using Eratosthenes' sieve

is 3 prime?

yes -- so cross out multiples of 3

Eratosthenes' Sieve

- is n prime? check by dividing n by each prime between 2 and $\sqrt{n}$
- list of primes between 2 and x can be calculated in linear time using Eratosthenes' sieve

Eratosthenes' Sieve

- is n prime? check by dividing n by each prime between 2 and $\sqrt{n}$
- list of primes between 2 and x can be calculated in linear time using Eratosthenes' sieve

is 4 prime?

no

```
public class HashTable {
    private List[] theTable;
.
.
.
}
```

```
public class HashTable {
.
.
.
    private static boolean prime (int n) {
        int sqrtN = (int) Math.sqrt(n);
        boolean[] sieve = new boolean[sqrtN];
        int i;
        for (i = 2; i < sqrtN; i++) sieve[i] = true;
        for (int p = 2; p < sqrtN; p++)
            if (sieve[p])
                for (i = 2 * p; i < sqrtN; i += p)
                    sieve[i] = false;
        for (i = 2; i < sqrtN; i++)
            if (sieve[i] && n % i == 0) return false;
        return true;
    }

    private static int nextPrime (int n) {
        while (true) if (prime(++n)) return n;
    }
.
.
.
}
```

```
public class HashTable {
.
.
.
    public HashTable (int estimatedElements) {
        int size = nextPrime(2 * estimatedElements);
        this.theTable = new List[size];
        for (int i = 0; i < size; i++)
            this.theTable[i] = List.NIL;
    }

    public int size () {
        return this.theTable.length;
    }
.
.
.
}
```

```
public class HashTable {
.
.
.
    public HashData find (HashData d) {
        int i = d.hashCode();
        List hit = this.theTable[i].find(d);
        return hit.isEmpty() ? null : (HashData) hit.head();
    }
.
.
.
}
```

```
public class HashTable {
.
.
.
    public void add (HashData d) {
        int i = d.hashCode();
        List hit = this.theTable[i].find(d);
        if (!hit.isEmpty()) ((HashData) hit.head()).incr();
        else this.theTable[i] = this.theTable[i].push(d);
    }
.
.
.
}
```

```
public class HashTable {
.
.
.
    public String toString () {
        String result = "";
        for (int i = 0; i < this.size(); i++)
            if (!this.theTable[i].isEmpty())
                result += i + ": " + this.theTable[i] + "\n";
        return result;
    }
.
.
.
}
```

```
public class HashTable {
.
.
    public void writeStats () {
        int longest = 0;
        int collisions = 0;
        int elements = 0;
        int empty = 0;
        for (int i = 0; i < this.size(); i++)
            if (!this.theTable[i].isEmpty()) {
                int len = this.theTable[i].length();
                longest = len > longest ? len : longest;
                collisions += len - 1;
                elements += len;
            } else empty++;
        IO.stdout.println((100*empty/theTable.length)
                          + "% empty, "
                          + elements + " elements, "
                          + collisions + " collisions, "
                          + longest + " longest, "
                          + ((float)elements /
                             (float)(this.size() - empty))
                          + " average.");
    }
}
```