13. Symbol Tables
13. Symbol Tables

- APIs and clients
- A design challenge
- Binary search trees
- Implementation
- Analysis
FAQs about sorting and searching

Hey, Alice. That whitelist filter with mergesort and binary search is working great.

Right, but it's a pain sometimes.

Why?

We have to sort the whole list whenever we add new customers.

Also, we want to process transactions and associate all sorts of information with our customers.

Bottom line. Need a more flexible API.
Why are telephone books obsolete?

Unsupported operations
• Change the number associated with a given name.
• Add a new name, associated with a given number.
• Remove a given name and associated number

Observation. Mergesort + binary search has the same problem with add and remove.

see Sorting and Searching lecture
Associative array abstraction

Imagine using arrays whose indices are *string* values.

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>(212) 123-4567</td>
</tr>
<tr>
<td>Bob</td>
<td>(609) 987-6543</td>
</tr>
<tr>
<td>Carl</td>
<td>(800) 888-8888</td>
</tr>
<tr>
<td>Dave</td>
<td>(888) 800-0800</td>
</tr>
<tr>
<td>Eve</td>
<td>(999) 999-9999</td>
</tr>
</tbody>
</table>

### A fundamental abstraction
- Use *keys* to access associated *values*.
- Keys and values could be any type of data.
- Client code could not be simpler.

### Q. How to implement?

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>transactions[Alice]</td>
<td>Dec 12 12:01AM $111.11 Amazon, Dec 12 1:11 AM $989.99 Ebay</td>
</tr>
<tr>
<td>transactions[Bob]</td>
<td></td>
</tr>
<tr>
<td>transactions[Carl]</td>
<td></td>
</tr>
<tr>
<td>transactions[Dave]</td>
<td></td>
</tr>
<tr>
<td>transactions[Eve]</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>URLs[128.112.136.11]</td>
<td><a href="http://www.cs.princeton.edu">www.cs.princeton.edu</a></td>
</tr>
<tr>
<td>URLs[128.112.128.15]</td>
<td><a href="http://www.princeton.edu">www.princeton.edu</a></td>
</tr>
<tr>
<td>URLs[130.132.143.21]</td>
<td><a href="http://www.yale.edu">www.yale.edu</a></td>
</tr>
<tr>
<td>URLs[128.103.060.55]</td>
<td><a href="http://www.harvard.edu">www.harvard.edu</a></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPAddrs[<a href="http://www.cs.princeton.edu">www.cs.princeton.edu</a>]</td>
<td>128.112.136.11</td>
</tr>
<tr>
<td>IPAddrs[<a href="http://www.princeton.edu">www.princeton.edu</a>]</td>
<td>128.112.128.15</td>
</tr>
<tr>
<td>IPAddrs[<a href="http://www.yale.edu">www.yale.edu</a>]</td>
<td>130.132.143.21</td>
</tr>
<tr>
<td>IPAddrs[<a href="http://www.harvard.edu">www.harvard.edu</a>]</td>
<td>128.103.060.55</td>
</tr>
</tbody>
</table>
Symbol table ADT

A symbol table is an ADT whose values are sets of key-value pairs, with keys all different.

Basic symbol-table operations

• Associate a given key with a given value.  
  [If the key is not in the table, add it to the table.]  
  [If the key is in the table, change its value.]
• Return the value associated with a given key.
• Test if a given key is in the table.
• Iterate through the keys.

Useful additional assumptions

• Keys are comparable and iteration is in order.
• No limit on number of key-value pairs.
• All keys not in the table associate with null.
Benchmark example of symbol-table operations

**Application.** Count frequency of occurrence of strings in StdIn.

**Keys.** Strings from a sequence.

**Values.** Integers.

<table>
<thead>
<tr>
<th>key</th>
<th>it</th>
<th>was</th>
<th>the</th>
<th>best</th>
<th>of</th>
<th>times</th>
<th>it</th>
<th>was</th>
<th>the</th>
<th>worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Parameterized API for symbol tables

**Goal.** Simple, safe, and clear client code for symbol tables holding any type of data.

Java approach: Parameterized data types (generics)
- Use placeholder type names for *both* keys and values.
- Substitute concrete types for placeholder in clients.

```
public class ST<Key extends Comparable<Key>, Value> {
    ST<Key, Value>()  // create a symbol table
    void put(Key key, Value val)  // associate key with val
    Value get(Key key)  // return value associated with key, null if none
    boolean contains(Key key)  // is there a value associated with key?
    Iterable<Key> keys()  // all the keys in the table
}
```

"implements compareTo()"
Aside: Iteration (client code)

Q. How to print the contents of a stack/queue?

A. Use Java's *foreach* construct.

**Enhanced for loop.**
- Useful for any collection.
- Iterate through each item in the collection.
- Order determined by implementation.
- Substantially simplifies client code.
- Works when API "implements Iterable".

**Java foreach construct**

```java
Stack<String> stack = new Stack<String>();
...
for (String s : stack)
    StdOut.println(s);
...
```

```
public class Stack<Item> {
    // implements Iterable<Item>
    Stack<Item>()
    void push(Item item)
    Item pop()
    boolean isEmpty()
    int size()
}
```

**Performance specification.** Constant-time per item.
Aside: Iteration (implementation)

Q. How to "implement Iterable"?

A. We did it for Stack and Queue, so you don't have to.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class Stack&lt;Item&gt;</td>
<td>implements Iterable&lt;Item&gt;</td>
</tr>
<tr>
<td>Stack&lt;Item&gt;()</td>
<td>create a stack of objects, all of type Item</td>
</tr>
<tr>
<td>void push(Item item)</td>
<td>add item to stack</td>
</tr>
<tr>
<td>Item pop()</td>
<td>remove and return item most recently pushed</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the stack empty?</td>
</tr>
<tr>
<td>int size()</td>
<td># of objects on the stack</td>
</tr>
</tbody>
</table>

A. Implement an Iterator (see text)

Meets performance specification. Constant-time per entry.

Bottom line. *Use iteration* in client code that uses collections.
**Why ordered keys?**

**Natural for many applications**
- Numeric types.
- Strings.
- Date and time.
- Client-supplied types (color, length).

**Enables useful API extensions**
- Provide the keys in sorted order.
- Find the $k$th smallest key.

**Enables efficient implementations**
- Mergesort.
- Binary search.
- BSTs (this lecture).
Symbol table client example 1: Sort (with dedup)

Goal. Sort lines on standard input (and remove duplicates).
• Key type. String (line on standard input).
• Value type. (ignored).

```java
public class Sort {
    public static void main(String[] args) {
        // Sort lines on StdIn
        BST<String, Integer> st = new BST<String, Integer>();
        while (StdIn.hasNextLine())
            st.put(StdIn.readLine(), 0);
        for (String s : st.keys())
            StdOut.println(s);
    }
}
```

% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
it was the spring of hope
it was the winter of despair

% java Sort < tale.txt
it was the age of foolishness
it was the age of wisdom
it was the best of times
it was the epoch of belief
it was the epoch of incredulity
it was the season of darkness
it was the season of light
it was the spring of hope
it was the winter of despair
it was the worst of times
Symbol table client example 2: Frequency counter

Goal. Compute frequencies of words on standard input.
• Key type. String (word on standard input).
• Value type. Integer (frequency count).

```java
public class Freq {
    public static void main(String[] args) {
        // Frequency counter
        BST<String, Integer> st = new BST<String, Integer>();
        while (!StdIn.isEmpty()) {
            String key = StdIn.readString();
            if (st.contains(key)) st.put(key, st.get(key) + 1);
            else st.put(key, 1);
        }
        for (String s : st.keys())
            StdOut.printf("%d %s\n", st.get(s), s);
    }
}
```

% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the belief of times
it was the despair of times
it was the foolishness of times
it was the wisdom of times
it was the epoch of times
it was the season of times
it was the spring of times
it was the winter of times
it was the youth of times
it was the age of times
it was the epoch of times
it was the season of times
it was the spring of times
it was the winter of times
it was the wisdom of times
it was the belief of times
it was the despair of times
it was the foolishness of times
```
Symbol table client example 3: Index

Goal. Print index to words on standard input.
- Key type. String (word on standard input).
- Value type. Queue<Integer> (indices where word occurs).

public class Index
{
    public static void main(String[] args)
    {
        BST<String, Queue<Integer>> st;
        st = new BST<String, Queue<Integer>>()
        for (int i = 0; !StdIn.isEmpty(); i++)
        {
            String key = StdIn.readString();
            if (!st.contains(key))
                st.put(key, new Queue<Integer>());
            st.get(key).enqueue(i);
        }
        for (String s : st.keys())
            StdOut.println(s + " " + st.get(s));
    }
}
Symbol-table applications

Symbol tables are *ubiquitous* in today's computational infrastructure.

We're going to need a good symbol-table implementation!

<table>
<thead>
<tr>
<th>application</th>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>contacts</td>
<td>name</td>
<td>phone number, address</td>
</tr>
<tr>
<td>credit card</td>
<td>account number</td>
<td>transaction details</td>
</tr>
<tr>
<td>file share</td>
<td>name of song</td>
<td>computer ID</td>
</tr>
<tr>
<td>dictionary</td>
<td>word</td>
<td>definition</td>
</tr>
<tr>
<td>web search</td>
<td>keyword</td>
<td>list of web pages</td>
</tr>
<tr>
<td>book index</td>
<td>word</td>
<td>list of page numbers</td>
</tr>
<tr>
<td>cloud storage</td>
<td>file name</td>
<td>file contents</td>
</tr>
<tr>
<td>domain name service</td>
<td>domain name</td>
<td>IP address</td>
</tr>
<tr>
<td>reverse DNS</td>
<td>IP address</td>
<td>domain name</td>
</tr>
<tr>
<td>compiler</td>
<td>variable name</td>
<td>value and type</td>
</tr>
<tr>
<td>internet routing</td>
<td>destination</td>
<td>best route</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

We're going to need a good symbol-table implementation!
Image sources

https://openclipart.org/detail/25617/astrid-graeber-adult-by-anonymous-25617
https://openclipart.org/detail/169320/girl-head-by-jza
https://www.flickr.com/photos/tunnelbug/8317946457/
http://www.socializedpr.com/unwanted-pre-information-era-phone-books-piling-up/
13. Symbol Tables

- APIs and clients
- A design challenge
- Binary search trees
- Implementation
- Analysis
**Benchmark**

**Application.** Linguistic analysis

**Zipf’s law** (for a natural language corpus)
- Suppose most frequent word occurs about $t$ times.
- 2nd most frequent word occurs about $t/2$ times.
- 3rd most frequent word occurs about $t/3$ times.
- 4th most frequent word occurs about $t/4$ times.

**Goal.** Validate Zipf’s law for real natural language data.

**Method.**

```
% java Freq < data.txt | java Sort
```

**Required.** Efficient symbol-table implementation.

---

**Example:**

<table>
<thead>
<tr>
<th>Word</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>the</td>
<td>13967</td>
</tr>
<tr>
<td>of</td>
<td>6415</td>
</tr>
<tr>
<td>and</td>
<td>6247</td>
</tr>
<tr>
<td>a</td>
<td>4583</td>
</tr>
<tr>
<td>to</td>
<td>4508</td>
</tr>
<tr>
<td>that</td>
<td>2911</td>
</tr>
<tr>
<td>his</td>
<td>2481</td>
</tr>
<tr>
<td>it</td>
<td>2370</td>
</tr>
<tr>
<td>i</td>
<td>1940</td>
</tr>
</tbody>
</table>
Benchmark statistics

**Goal.** Validate Zipf's law for real natural language data.

**Method.**

```
% java Freq < data.txt | java Sort
```

<table>
<thead>
<tr>
<th>file</th>
<th>description</th>
<th>words</th>
<th>distinct</th>
</tr>
</thead>
<tbody>
<tr>
<td>mobydickek.txt</td>
<td>Melville's <em>Moby Dick</em></td>
<td>210,028</td>
<td>16,834</td>
</tr>
<tr>
<td>liepzig100k.txt</td>
<td>100K random sentences</td>
<td>2,121,054</td>
<td>144,256</td>
</tr>
<tr>
<td>liepzig200k.txt</td>
<td>200K random sentences</td>
<td>4,238,435</td>
<td>215,515</td>
</tr>
<tr>
<td>liepzig1m.txt</td>
<td>1M random sentences</td>
<td>21,191,455</td>
<td>534,580</td>
</tr>
</tbody>
</table>

Reference: Wortschatz corpus, Universität Leipzig
http://corpora.informatik.uni-leipzig.de

**Required.** Efficient symbol-table implementation.
### Strawman I: Ordered array

#### Idea
- Keep keys in order in an array.
- Keep values in a parallel array.

#### Reasons (see "Sorting and Searching" lecture)
- Takes advantage of fast sort (mergesort).
- Enables fast search (binary search).

#### Known challenge. How big to make the arrays?

#### Fatal flaw. How to insert a new key?
- To keep key array in order, need to move larger entries à la insertion sort.
- Hypothesis: Quadratic time for benchmark.
Strawman II: Linked list

Idea
- Keep keys in order in a linked list.
- Add a value to each node.


Fatal flaw. How to search?
- Binary search requires indexed access.
- Example: How to access the middle of a linked list?
- Only choice: search `sequentially` through the list.
- Hypothesis: Quadratic time for benchmark.

easy to validate with experiments
**Design challenge**

Implement **scalable** symbol tables.

**Goal.** Simple, safe, clear, and **efficient** client code.

**Performance specifications**

- Order of growth of running time for put(), get() and contains() is **logarithmic**.
- Memory usage is linear in the size of the collection, when it is nonempty.
- No limits within the code on the collection size.

**Are such guarantees achievable??**

Can we implement associative arrays with just log-factor extra cost??

```
phoneNumbers["Alice"] = "(212) 123-4567"
```

*Only slightly more costly than stacks or queues!*

*No way!*
13. Symbol Tables

- APIs and clients
- A design challenge
- Binary search trees
- Implementation
- Analysis
Doubly-linked data structures

With two links ($\bigcirc \leftarrow \bigcirc$) a wide variety of data structures are possible.

Doubly-linked list

Doubly-linked circular list

Maintenance can be complicated!

Binary tree (this lecture)

Tree

General case

From the point of view of a particular object, all of these structures look the same.
A doubly-linked data structure: binary search tree

Binary search tree (BST)
- A recursive data structure containing distinct comparable keys that is *ordered*.
- **Def.** A *BST* is a null or a reference to a *BST node* (the root).
- **Def.** A *BST node* is a data type that contains references to a key, a value, and two BSTs, a *left* subtree and a *right* subtree.
- **Ordered.** All keys in the *left* subtree of each node are *smaller* than its key and all keys in the *right* subtree of each node are *larger* than its key.

```java
private class Node {
    private Key key;
    private Value val;
    private Node left;
    private Node right;
    public Node(Key key, Value val) {
        this.key = key; this.val = val;
    }
}
```

A BST
Standard operations for processing data structured as a binary search tree

- Search for the value associated with a given key.
- Add a new key-value pair.
- Traverse the BST (visit every node, in order of the keys).
- Remove a given key and associated value (not addressed in this lecture).
**BST processing code: Search**

**Goal.** Find the value associated with a given key in a BST.
- If *less* than the key at the current node, go *left*.
- If *greater* than the key at the current node, go *right*.

**Example.** `get("the")`

```
public Value get(Key key)
{
    return get(root, key);
}

private Value get(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return get(x.left, key);
    else if (cmp > 0) return get(x.right, key);
    else return x.val;
}
```
**BST processing code: Associate a new value with a key**

**Goal.** Associate a new value with a given key in a BST.
- If *less* than the key at the current node, go *left*.
- If *greater* than the key at the current node, go *right*.

**Example.** put("the", 2) root

```
public void put(Key key, Value val)
{  root = put(root, key, val);  }
private Node put(Node x, Key key, Value val)
{  if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if    (cmp < 0) x.left = put(x.left, key, val);
   else if (cmp > 0) x.right = put(x.right, key, val);
   else    x.val = val;
   return x;
}
```
**Goal.** Add a new key-value pair to a BST.
- Search for key.
- Return link to new node when *null* reached.

**Example.**
```java
class Node {
    int key;
    Object val;
    Node left, right;
}

class BST {
    Node root;
	public void put(Key key, Value val) {
	    root = put(root, key, val);
    }
}
```

```java
public void put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = put(x.left, key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else x.val = val;
    return x;
}
```
## BST processing code: Traverse the BST

**Goal.** Put keys in a BST on a queue, in sorted order.
- Do it for the left subtree.
- Put the key at the root on the queue.
- Do it for the right subtree.

```java
public Iterable<Key> keys() {
    Queue<Key> queue = new Queue<Key>();
    inorder(root, queue);
    return queue;
}

private void inorder(Node x, Queue<Key> queue) {
    if (x == null) return;
    inorder(x.left, queue);
    queue.enqueue(x.key);
    inorder(x.right, queue);
}
```

```
Queue [ best | it | of | the | times | was ]
```
CS.13.C.SymbolTables.BSTs
13. Symbol Tables

- APIs and clients
- A design challenge
- Binary search trees
- Implementation
- Analysis
ADT for symbol tables: review

A symbol table is an idealized model of an associative storage mechanism.

An ADT allows us to write Java programs that use and manipulate symbol tables.

```java
public class ST<Key extends Comparable<Key>, Value> {
    ST<Key, Value>()
    void put(Key key, Value val)
    Value get(Key key)
    boolean contains(Key key)
    Iterable<Key> keys()
}
```

Performance specifications

- Order of growth of running time for put(), get() and contains() is logarithmic.
- Memory usage is linear in the size of the collection, when it is nonempty.
- No limits within the code on the collection size.
Data structure choice. Use a BST to hold the collection.

```java
public class BST<Key extends Comparable<Key>, Value> {
    private Node root = null;

    private class Node {
        private Key key;
        private Value val;
        private Node left;
        private Node right;
        public Node(Key key, Value val) {
            this.key = key;
            this.val = val;
        }
    }
    ...
}
```
BST implementation: Test client (frequency counter)

```java
public static void main(String[] args)
{
    BST<String, Integer> st = new BST<String, Integer>();
    while (!StdIn.isEmpty())
    {
        String key = StdIn.readString();
        if (st.contains(key)) st.put(key, st.get(key) + 1);
        else st.put(key, 1);
    }
    for (String s : st.keys())
    {
        StdOut.printf("%d %s\n", st.get(s), s);
    }
}
```

What we expect, once the implementation is done.

```
% java BST < tale.txt
  2 age
  1 belief
  1 best
  1 darkness
  1 despair
  2 epoch
  1 foolishness
  1 hope
  1 incredulity
 10 it
  1 light
 10 of
  2 season
  1 spring
 10 the
  2 times
 10 was
  1 winter
  1 wisdom
  1 worst
```
BST implementation: Methods

**Methods** define data-type operations (implement the API).

```java
public class BST<Key extends Comparable<Key>, Value> {
    ...

    public boolean isEmpty() 
    { return root == null; }

    public void put(Key key, Value value) 
    { /* See BST add slides and next slide. */ }

    public Value get(Key key) 
    { /* See BST search slide and next slide. */ }

    public boolean contains(Key key) 
    { return get(key) != null; }

    public Iterable<Key> keys() 
    { /* See BST traverse slide and next slide. */ }

    ...
}
```
BST implementation

```java
public class BST<Key extends Comparable<Key>, Value> {
    private Node root = null;

    private class Node {
        private Key key;
        private Value val;
        private Node left;
        private Node right;
        private Node(Key key, Value val) {
            this.key = key; this.val = val;
        }
    }

    public boolean isEmpty() {
        return root == null;
    }

    public void put(Key key, Value val) {
        root = put(root, key, val);
    }

    public Value get(Key key) {
        return get(root, key);
    }

    public boolean contains(Key key) {
        return get(key) != null;
    }

    public Iterable<Key> keys() {
        Queue<Key> queue = new Queue<Key>();
        inorder(root, queue);
        return queue;
    }

    private Value get(Node x, Key key) {
        if (x == null) return null;
        int cmp = key.compareTo(x.key);
        if (cmp < 0) return get(x.left, key);
        else if (cmp > 0) return get(x.right, key);
        else return x.val;
    }

    private Node put(Node x, Key key, Value val) {
        if (x == null) return new Node(key, val);
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x.left = put(x.left, key, val);
        else if (cmp > 0) x.right = put(x.right, key, val);
        else x.val = val;
        return x;
    }

    private void inorder(Node x, Queue<Key> queue) {
        if (x == null) return;
        inorder(x.left, queue);
        q.enqueue(x.key);
        inorder(x.right, queue);
    }

    public static void main(String[] args) {
        /* Frequency counter */
    }
}
```
Trace of BST construction
13. Symbol Tables

- APIs and clients
- A design challenge
- Binary search trees
- Implementation
- Analysis
Costs depend on order of key insertion.

**Best case**

- `it` -> `best`
- `of` -> `times`
- `was` -> `worst`

**Worst case**

- `best` -> `it`
- `of` -> `worst`
- `times` -> `the`

**Typical case**

- `it` -> `best`
- `of` -> `times`
- `the` -> `worst`
Insert keys in random order.
- Tree is roughly balanced.
- Tends to stay that way!
**BST analysis**

**Model.** Insert keys in random order.
- Tree is roughly balanced.
- Tends to stay that way!

**Proposition.** Building a BST by inserting $N$ randomly ordered keys into an initially empty tree uses $\sim 2N \ln N$ (about $1.39 N \lg N$) compares.

**Proof.** A very interesting exercise in discrete math.
Benchmarking the BST implementation

**BST implements the associative-array abstraction** for randomly ordered keys.

```
public class ST<Key extends Comparable<Key>, Value> {
    ST<Key, Value> ()
        create a symbol table
    void put(Key key, Value value)
        associate key with value
    Value get(Key key)
        return value associated with key, null if none
    boolean contains(Key key)
        is there a value associated with key?
    Iterable<Key> keys()
        all the keys in the table (sorted)
}
```

**Symbol table API**

- Order of growth of running time for put(), get() and contains() is logarithmic.
- Memory use is linear in the size of the collection, when it is nonempty.
- No limits within the code on the collection size.

Made possible by *binary tree data structure.*
Empirical tests of BSTs

Count number of words that appear more than once in StdIn.

Frequency count without the output (DupsBST.java)

<table>
<thead>
<tr>
<th>( N )</th>
<th>( T_N ) (seconds)</th>
<th>( T_N / T_{N/2} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 million</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>2 million</td>
<td>9</td>
<td>1.8</td>
</tr>
<tr>
<td>4 million</td>
<td>17</td>
<td>1.9</td>
</tr>
<tr>
<td>8 million</td>
<td>34</td>
<td>2</td>
</tr>
<tr>
<td>16 million</td>
<td>72</td>
<td>2.1</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 BILLION</td>
<td>4608</td>
<td>2</td>
</tr>
</tbody>
</table>

\% java Generator 1000000 ... 263934 (5 seconds)
\% java Generator 2000000 ... 593973 (9 seconds)
\% java Generator 4000000 ... 908795 (17 seconds)
\% java Generator 8000000 ... 996961 (34 seconds)
\% java Generator 16000000 ... 999997 (72 seconds)

... = 6 0123456789 | java DupsBST

6-digit integers

Conforms hypothesis that order of growth is \( N \log N \)

WILL scale

Easy to process 21M word corpus NOT possible with brute-force
**Performance guarantees**

**Practical problem.** Keys may *not* be randomly ordered.
- BST may become unbalanced.
- Running time may be quadratic.
- Happens in practice (insert keys in order).

**Remarkable resolution.**
- *Balanced tree* algorithms perform simple transformations that *guarantee* balance.
- *Red-black trees* (Guibas and Sedgewick, 1979) are implemented in many modern systems.
Red-black tree insertion: random order visualization

Insert keys in random order.
• Same # of black links on every path from root to leaf.
• No two red links in a row.
• Tree is nearly balanced.
• Guaranteed to stay that way!
ST implementation with guaranteed logarithmic performance

```java
import java.util.TreeMap;

public class ST<Key extends Comparable<Key>, Value> {
    private TreeMap<Key, Value> st = new TreeMap<Key, Value>();

    public void put(Key key, Value val) {
        if (val == null) st.remove(key);
        else st.put(key, val);
    }

    public Value get(Key key) { return st.get(key); }
    public Value remove(Key key) { return st.remove(key); }
    public boolean contains(Key key) { return st.containsKey(key); }
    public Iterable<Key> keys() { return st.keySet(); }
}
```

Java's TreeMap library uses red-black trees.

**Proposition.** In a red-black tree of size $N$, put(), get() and contains() are **guaranteed** to use fewer than $2\lg N$ compares.

**Proof.** A fascinating exercise in algorithmics.

Interested in details? Take a course in algorithms.

Several other useful operations also available.
Summary

**BSTs.** Simple symbol-table implementation, usually efficient.

**Hashing.** More complicated symbol-table implementation, can be efficient.

**Red-black trees.** Variation of BSTs, *guaranteed* to be efficient.

**Applications.** Many, many, many things are enabled by efficient symbol tables.

---

Can we implement associative arrays with just log-factor extra cost??

---

**Example.** Search among 1 trillion customers with less than 80 compares (!)

**Example.** Search among all the atoms in the universe with less than 200 compares (!!!)
13. Symbol Tables