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.

- 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?

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.

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.

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>it</th>
<th>was</th>
<th>the best</th>
<th>of</th>
<th>times</th>
<th>it</th>
<th>was</th>
<th>the worst</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>1</td>
<td>i</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>i</td>
<td>1</td>
<td>it</td>
<td>best</td>
<td>best</td>
<td>best</td>
<td>best</td>
<td>best</td>
<td>best</td>
</tr>
<tr>
<td></td>
<td>it</td>
<td>1</td>
<td>it</td>
<td>of</td>
<td>of</td>
<td>of</td>
<td>of</td>
<td>of</td>
<td>of</td>
</tr>
<tr>
<td></td>
<td>was</td>
<td>1</td>
<td>the</td>
<td>times</td>
<td>times</td>
<td>times</td>
<td>times</td>
<td>times</td>
<td>worst</td>
</tr>
</tbody>
</table>

Symbol Table API

<table>
<thead>
<tr>
<th>public class ST&lt;Key extends Comparable&lt;Key&gt;, Value&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST&lt;Key, Value&gt;()</td>
</tr>
<tr>
<td>void put(Key key, Value val)</td>
</tr>
<tr>
<td>Value get(Key key)</td>
</tr>
<tr>
<td>boolean contains(Key key)</td>
</tr>
<tr>
<td>Iterable&lt;Key&gt; keys()</td>
</tr>
</tbody>
</table>
Aside: Iteration (client code)

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

A. Use Java’s `for each` 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’.

<table>
<thead>
<tr>
<th>Java foreach construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack&lt;String&gt; stack = new Stack&lt;String&gt;();</td>
</tr>
<tr>
<td>for (String s : stack)</td>
</tr>
<tr>
<td>StdOut.println(s);</td>
</tr>
</tbody>
</table>

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.

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?

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.

```
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);
    }
}
```
### 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("%8d %s\n", st.get(s), s);
    }
}
```

### 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).

```java
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.put(key, new Queue<Integer>());
            }
            for (String s : st.keys())
                StdOut.println(s + " " + st.get(s));
        }
    }
}
```

### Symbol table applications

Symbol tables are ubiquitous in today’s computational infrastructure.

<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>
</tbody>
</table>

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

**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>mobydict.txt</td>
<td>Melville’s Moby Dick</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>liepzig.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

**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.**

**easy to validate with experiments**
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?
• Hypothesis: Quadratic time for benchmark.

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??

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

This lecture. Yes way!

Only slightly more costly than stacks or queues!
Doubly-linked data structures

With two links (🔗) a wide variety of data structures are possible.

Maintenance can be complicated!

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; }
}
```

BST processing code

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")

```java
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.

```java
public void put(Key key, Value val) {
    Node x = root;
    while (x != null) {
        if (cmp(key, x.key) < 0) x = x.left;
        else if (cmp(key, x.key) > 0) x = x.right;
        else { x.val = val; return; }
    }
    x = new Node(key, val); // Add new node
    if (cmp(key, root.key) < 0) root.left = x;
    else root.right = x;
}
```

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

```
public void put(Key key, Value val) {
    Node x = root;
    while (x != null) {
        if (cmp(key, x.key) < 0) x = x.left;
        else if (cmp(key, x.key) > 0) x = x.right;
        else { x.val = val; return; }
    }
    x = new Node(key, val); // Add new node
    if (cmp(key, root.key) < 0) root.left = x;
    else root.right = x;
}
```

**BST processing code: Add a new key**

**Goal.** Add a new key-value pair to a BST.
- Search for key.
- Return link to new node when null reached.

```java
public void put(Key key, Value val) {
    Node x = root;
    while (x != null) {
        if (cmp(key, x.key) < 0) x = x.left;
        else if (cmp(key, x.key) > 0) x = x.right;
        else { x.val = val; return; }
    }
    x = new Node(key, val); // Add new node
    if (cmp(key, root.key) < 0) root.left = x;
    else root.right = x;
}
```

**Example.** put("worst", 1) root

```
public void put(Key key, Value val) {
    Node x = root;
    while (x != null) {
        if (cmp(key, x.key) < 0) x = x.left;
        else if (cmp(key, x.key) > 0) x = x.right;
        else { x.val = val; return; }
    }
    x = new Node(key, val); // Add new node
    if (cmp(key, root.key) < 0) root.left = x;
    else root.right = 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;
}
```

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

**Example.** put("important", 3) root
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.

<table>
<thead>
<tr>
<th>API</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>create symbol table</td>
<td>ST&lt;Key, Value&gt;()</td>
</tr>
<tr>
<td>associate key with val</td>
<td>void put(key, Value)</td>
</tr>
<tr>
<td>return value associated with key, null if none</td>
<td>Value get(key)</td>
</tr>
<tr>
<td>is there a value associated with key?</td>
<td>boolean contains(Key)</td>
</tr>
<tr>
<td>all the keys in the table</td>
<td>Iterable&lt;Key&gt; keys()</td>
</tr>
</tbody>
</table>

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.

Symbol table implementation: Instance variables and constructor

Data structure choice. Use a BST to hold the collection.

```
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)

```
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
", st.get(s), s);
}
```

What we expect, once the implementation is done.
**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. */
    }
    // ...
}
```

**Trace of BST construction**

```java
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;
}
```

```java
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;
}
```

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

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

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

**BST analysis**

Costs depend on order of key insertion.

**Best case**

**Worst case**

**Typical case**

---

**BST insertion: random order visualization**

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 $-2N \ln N$ (about $1.39N \lg N$) compares.

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

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

```java
public class ST<Key extends Comparable<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?
        }
    }
}
```

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.

Performance specifications

- Made possible by binary tree data structure.

Empirical tests of BSTs

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

<table>
<thead>
<tr>
<th>N</th>
<th>T[N] (seconds)</th>
<th>T[N]/T[½N]</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>

Frequency count without the output (DupsBST.java)

Confirms hypothesis that order of growth is $N \log N$

WILL scale

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 $2\log N$ compares.

Proof. A fascinating exercise in algorithmics.

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