3.1 Symbol Tables

- API
- elementary implementations
- ordered operations
“Smart data structures and dumb code works a lot better than the other way around.” — Eric S. Raymond
3.1 **Symbol Tables**

- **API**
  - *elementary implementations*
  - *ordered operations*
Why are telephone books obsolete?

Unsupported operations.

- Add a new name and associated number.
- Remove a given name and associated number.
- Change the number associated with a given name.
Symbol tables

Key–value pair abstraction.

- **Insert** a value with specified key.
- Given a key, **search** for the corresponding value.

**Ex.** DNS lookup.

- Insert domain name with specified IP address.
- Given domain name, find corresponding IP address.

<table>
<thead>
<tr>
<th>domain name</th>
<th>IP address</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.cs.princeton.edu">www.cs.princeton.edu</a></td>
<td>128.112.136.11</td>
</tr>
<tr>
<td><a href="http://www.princeton.edu">www.princeton.edu</a></td>
<td>128.112.128.15</td>
</tr>
<tr>
<td><a href="http://www.yale.edu">www.yale.edu</a></td>
<td>130.132.143.21</td>
</tr>
<tr>
<td><a href="http://www.harvard.edu">www.harvard.edu</a></td>
<td>128.103.060.55</td>
</tr>
</tbody>
</table>
# Symbol table applications

<table>
<thead>
<tr>
<th>application</th>
<th>purpose of search</th>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>dictionary</td>
<td>find definition</td>
<td>word</td>
<td>definition</td>
</tr>
<tr>
<td>book index</td>
<td>find relevant pages</td>
<td>term</td>
<td>list of page numbers</td>
</tr>
<tr>
<td>file share</td>
<td>find song to download</td>
<td>name of song</td>
<td>computer ID</td>
</tr>
<tr>
<td>financial account</td>
<td>process transactions</td>
<td>account number</td>
<td>transaction details</td>
</tr>
<tr>
<td>web search</td>
<td>find relevant web pages</td>
<td>keyword</td>
<td>list of page names</td>
</tr>
<tr>
<td>compiler</td>
<td>find properties of variables</td>
<td>variable name</td>
<td>type and value</td>
</tr>
<tr>
<td>routing table</td>
<td>route Internet packets</td>
<td>destination</td>
<td>best route</td>
</tr>
<tr>
<td>DNS</td>
<td>find IP address</td>
<td>domain name</td>
<td>IP address</td>
</tr>
<tr>
<td>reverse DNS</td>
<td>find domain name</td>
<td>IP address</td>
<td>domain name</td>
</tr>
<tr>
<td>genomics</td>
<td>find markers</td>
<td>DNA string</td>
<td>known positions</td>
</tr>
<tr>
<td>file system</td>
<td>find file on disk</td>
<td>filename</td>
<td>location on disk</td>
</tr>
</tbody>
</table>
Symbol tables: context

Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and \( n - 1 \).

Language support.

- External libraries: C, VisualBasic, Standard ML, bash, ...
- Built-in libraries: Java, C#, C++, Scala, ...
- Built-in to language: Awk, Perl, PHP, Tcl, JavaScript, Python, Ruby, Lua.

```
has_nice_syntax_for_associative_arrays["Python"] = True
has_nice_syntax_for_associative_arrays["Java"] = False
```

legal Python code
Basic symbol table API

**Associative array abstraction.** Associate one value with each key.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ST()</code></td>
<td>create an empty symbol table</td>
</tr>
<tr>
<td><code>void put(Key key, Value val)</code></td>
<td>insert key–value pair</td>
</tr>
<tr>
<td><code>Value get(Key key)</code></td>
<td>value paired with key</td>
</tr>
<tr>
<td><code>boolean contains(Key key)</code></td>
<td>is there a value paired with key?</td>
</tr>
<tr>
<td><code>Iterable&lt;Key&gt; keys()</code></td>
<td>all the keys in the symbol table</td>
</tr>
<tr>
<td><code>void delete(Key key)</code></td>
<td>remove key (and associated value)</td>
</tr>
<tr>
<td><code>boolean isEmpty()</code></td>
<td>is the symbol table empty?</td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>number of key–value pairs</td>
</tr>
</tbody>
</table>

```java
da[key] = val;
da[key]
```
Conventions

- Method `get()` returns `null` if key not present.
- Method `put()` overwrites old value with new value.
- Values are not `null`.  

```
java.util.Map allows null values
```

“Careless use of `null` can cause a staggering variety of bugs. Studying the Google code base, we found that something like 95% of collections weren't supposed to have any `null` values in them, and having those fail fast rather than silently accept `null` would have been helpful to developers.”

[guava-libraries](https://code.google.com/p/guava-libraries/wiki/UsingAndAvoidingNullExplained)
Conventions

- Method `get()` returns `null` if key not present.
- Method `put()` overwrites old value with new value.
- Values are not `null`.

**Intended consequences.**

- Easy to implement `contains()`.

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

- Can implement lazy version of `delete()`.

  ```java
  public void delete(Key key) {
      put(key, null);
  }
  ```
Keys and values

Value type. Any generic type.

Key type: several natural assumptions.

- Assume keys are Comparable, use compareTo().
- Assume keys are any generic type, use equals() to test equality.
- Assume keys are any generic type, use equals() to test equality; use hashCode() to scramble key.

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: Integer, Double, String, java.io.File, ...
- Mutable in Java: StringBuilder, java.net.URL, arrays, ...
Equality test

All Java classes inherit a method `equals()`.

**Java requirements.** For any references `x`, `y` and `z`:

- Reflexive: `x.equals(x)` is true.
- Symmetric: `x.equals(y)` iff `y.equals(x)`.
- Transitive: if `x.equals(y)` and `y.equals(z)`, then `x.equals(z)`.
- Non-null: `x.equals(null)` is false.

**Default implementation.** `(x == y)

**Customized implementations.** Integer, Double, String, java.io.File, ...

**User-defined implementations.** Some care needed.
Implementing equals for user-defined types

Seems easy.

```java
public class Date implements Comparable<Date> {
    private final int month;
    private final int day;
    private final int year;
    ... 
    public boolean equals(Date that) {
        if (this.day != that.day) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year) return false;
        return true;
    }
}
```

check that all significant fields are the same
Implementing equals for user-defined types

Seems easy, but requires some care.

```java
public final class Date implements Comparable<Date>
{
    private final int month;
    private final int day;
    private final int year;
    
    public boolean equals(Object y)
    {
        if (y == this) return true;
        if (y == null) return false;

        Date that = (Date) y;
        if (this.day != that.day ) return false;
        if (this.month != that.month) return false;
        if (this.year != that.year ) return false;
        return true;
    }
}
```

typically unsafe to use equals() with inheritance (would violate symmetry)
must be Object. Why? Experts still debate.
optimization (for reference equality)
check for null
objects must be in the same class (religion: getClass() vs. instanceof)
cast is now guaranteed to succeed
check that all significant fields are the same
Equals design

“Standard” recipe for user-defined types.

- Optimization for reference equality.
- Check against `null`.
- Check that two objects are of the same type; cast.
- Compare each significant field:
  - if field is a primitive type, use `==`
  - if field is an object, use `equals()`
  - if field is an array, apply to each entry

Best practices.

- No need to use calculated fields that depend on other fields.
- Compare fields mostly likely to differ first.
- Make `compareTo()` consistent with `equals()`.

\[ x.equals(y) \text{ if and only if } (x.compareTo(y) == 0) \]
ST test client for analysis

**Frequency counter.** Read a sequence of strings from standard input; print one that occurs most often.

```bash
% more tinyTale.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
```

```bash
% java FrequencyCounter 3 < tinyTale.txt
the 10
```

```bash
% java FrequencyCounter 8 < tale.txt
business 122
```

```bash
% java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

tiny example (60 words, 20 distinct)

real example (135,635 words, 10,769 distinct)

real example (21,191,455 words, 534,580 distinct)
public class FrequencyCounter {
    public static void main(String[] args) {
        int minLength = Integer.parseInt(args[0]);

        ST<String, Integer> st = new ST<String, Integer>();
        while (!StdIn.isEmpty()) {
            String word = StdIn.readString();
            if (word.length() < minLength) continue;
            if (!st.contains(word)) st.put(word, 1);
            else st.put(word, st.get(word) + 1);
        }

        String max = "";
        st.put(max, 0);
        for (String word : st.keys()) {
            if (st.get(word) > st.get(max))
                max = word;
        }
        StdOut.println(max + " " + st.get(max));
    }
}
3.1 Symbol Tables

- API
- elementary implementations
- ordered operations
Sequential search in a linked list

**Data structure.** Maintain an (unordered) linked list of key–value pairs.

**Search.** Scan through all keys until find a match.

**Insert.** Scan through all keys until find a match; if no match add to front.

get("A")

\[
\text{X}7 \rightarrow \text{H}5 \rightarrow \text{C}4 \rightarrow \text{R}3 \rightarrow \text{A}8 \rightarrow \text{E}6 \rightarrow \text{S}0
\]

put("M", 9)

\[
\text{M}9 \rightarrow \text{X}7 \rightarrow \text{H}5 \rightarrow \text{C}4 \rightarrow \text{R}3 \rightarrow \text{A}8 \rightarrow \text{E}6 \rightarrow \text{S}0
\]
Elementary ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
</tbody>
</table>
Binary search in an ordered array

**Data structure.** Maintain parallel arrays for keys and values, sorted by keys.

**Search.** Use binary search to find key.

**Proposition.** At most $\sim \log n$ compares to search a sorted array of length $n$.

---

**get("P")**

<table>
<thead>
<tr>
<th>keys[]</th>
<th>vals[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>A C E H L M P R S X</td>
<td>8 4 2 5 11 9 10 3 0 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>lo hi m</th>
<th>entries in black are a[lo..hi]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 9 4</td>
<td>A C E H L M P R S X</td>
</tr>
<tr>
<td>5 9 7</td>
<td>A C E H L M P R S X</td>
</tr>
<tr>
<td>5 6 5</td>
<td>A C E H L M P R S X</td>
</tr>
<tr>
<td>6 6 6</td>
<td>A C E H L M P R S X</td>
</tr>
</tbody>
</table>

entry in red is a[m]

return vals[6]
Binary search in an ordered array

**Data structure.** Maintain parallel arrays for keys and values, sorted by keys.

**Search.** Use binary search to find key.

```java
public Value get(Key key) {
    int lo = 0, hi = n-1;
    while (lo <= hi) {
        int mid = lo + (hi - lo) / 2;
        int cmp = key.compareTo(keys[mid]);
        if (cmp < 0) hi = mid - 1;
        else if (cmp > 0) lo = mid + 1;
        else if (cmp == 0) return vals[mid];
    }
    return null;  // no matching key
}
```
Implementing binary search was

A. Much easier than I thought.
B. Easier than I thought.
C. About what I expected.
D. Harder than I thought.
E. Much harder than I thought.
**Problem.** Given an array with all 0s in the beginning and all 1s at the end, with more 1s than 0s, find the index in the array where the 1s begin.
Binary search: insert

Data structure. Maintain an ordered array of key–value pairs.

Insert. Use binary search to find place to insert; shift all larger keys over.

Proposition. Takes linear time in the worst case.

put("P", 10)

<table>
<thead>
<tr>
<th>keys[]</th>
<th>vals[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>0 1 2 3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>A C E H M R S X - -</td>
<td>8 4 6 5 9 3 0 7 - -</td>
</tr>
</tbody>
</table>
## Elementary ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>$n$</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\log n$</td>
<td>$n$†</td>
<td>$\log n$</td>
</tr>
</tbody>
</table>

† can do with $\log n$ compares, but requires $n$ array accesses

### Challenge.
Efficient implementations of both search and insert.
3.1 Symbol Tables

- API
- elementary implementations
- ordered operations
Examples of ordered symbol table API

<table>
<thead>
<tr>
<th>keys</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>min()</td>
<td>09:00:00 Chicago</td>
</tr>
<tr>
<td></td>
<td>09:00:03 Phoenix</td>
</tr>
<tr>
<td></td>
<td>09:00:13 Houston</td>
</tr>
<tr>
<td>get(09:00:13)</td>
<td>09:00:59 Chicago</td>
</tr>
<tr>
<td></td>
<td>09:01:10 Houston</td>
</tr>
<tr>
<td>floor(09:05:00)</td>
<td>09:03:13 Chicago</td>
</tr>
<tr>
<td></td>
<td>09:10:11 Seattle</td>
</tr>
<tr>
<td>select(7)</td>
<td>09:10:25 Seattle</td>
</tr>
<tr>
<td></td>
<td>09:14:25 Phoenix</td>
</tr>
<tr>
<td></td>
<td>09:19:32 Chicago</td>
</tr>
<tr>
<td></td>
<td>09:19:46 Chicago</td>
</tr>
<tr>
<td>keys(09:15:00, 09:25:00)</td>
<td>09:21:05 Chicago</td>
</tr>
<tr>
<td></td>
<td>09:22:43 Seattle</td>
</tr>
<tr>
<td></td>
<td>09:22:54 Seattle</td>
</tr>
<tr>
<td></td>
<td>09:25:52 Chicago</td>
</tr>
<tr>
<td>ceiling(09:30:00)</td>
<td>09:35:21 Chicago</td>
</tr>
<tr>
<td></td>
<td>09:36:14 Seattle</td>
</tr>
<tr>
<td>max()</td>
<td>09:37:44 Phoenix</td>
</tr>
</tbody>
</table>

size(09:15:00, 09:25:00) is 5
rank(09:10:25) is 7
Ordered symbol table API

```java
class ST<Key extends Comparable<Key>, Value> {
    // Method signatures
    Key min(); // smallest key
    Key max(); // largest key
    Key floor(Key key); // largest key less than or equal to key
    Key ceiling(Key key); // smallest key greater than or equal to key
    int rank(Key key); // number of keys less than key
    Key select(int k); // key of rank k
    // ...
}
```
**Problem.** Given a sorted array of $n$ distinct keys, find the number of keys strictly less than a given query key.
## Binary search: ordered symbol table operations summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sequential search</th>
<th>Binary search</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>insert</td>
<td>$n$</td>
<td>$n$</td>
</tr>
<tr>
<td>min / max</td>
<td>$n$</td>
<td>1</td>
</tr>
<tr>
<td>floor / ceiling</td>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>rank</td>
<td>$n$</td>
<td>$\log n$</td>
</tr>
<tr>
<td>select</td>
<td>$n$</td>
<td>1</td>
</tr>
</tbody>
</table>

**Order of growth of the running time for ordered symbol table operations**