Symbol tables

Key-value pair abstraction.
- Insert a value with specified key.
- Given a key, search for the corresponding value.

Ex. DNS lookup.
- Insert URL with specified IP address.
- Given URL, find corresponding IP address.

<table>
<thead>
<tr>
<th>URL</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>130.143.143.21</td>
</tr>
<tr>
<td><a href="http://www.harvard.edu">www.harvard.edu</a></td>
<td>128.103.040.15</td>
</tr>
<tr>
<td><a href="http://www.simpsons.com">www.simpsons.com</a></td>
<td>209.052.165.60</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 given URL</td>
<td>URL</td>
<td>IP address</td>
</tr>
<tr>
<td>reverse DNS</td>
<td>find URL given IP address</td>
<td>IP address</td>
<td>URL</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>

Conventions

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

Intended consequences.
- Easy to implement contains().

```java
class BasicSymbolTable {
    private static int[] a;
    public boolean contains(Key key) {
        return get(key) != null;
    }
    public void delete(Key key) {
        put(key, null);
    }
}
```

- Can implement lazy version of delete().

```java
class BasicSymbolTable {
    private static int[] a;
    public boolean contains(Key key) {
        return get(key) != null;
    }
    public void delete(Key key) {
        put(key, null);
    }
}
```

Basic symbol table API

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

### public class ST<Key, Value>

- **create a symbol table**
- **put(key, value)** (remove key from table if value is null)
- **get(key)** (value paired with key (null if key is absent))
- **delete(key)** (remove key and its value from table)
- **contains(key)** (is there a value paired with key?)
- **isEmpty()** (is the table empty?)
- **size()** (number of key-value pairs in the table)
- **Iterable<Key>** **keys()** (all the keys in the table)

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

**Conventions**

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

**Intended consequences.**
- Easy to implement contains().

```java
class BasicSymbolTable {
    private static int[] a;
    public boolean contains(Key key) {
        return get(key) != null;
    }
    public void delete(Key key) {
        put(key, null);
    }
}
```

- Can implement lazy version of delete().

```java
class BasicSymbolTable {
    private static int[] a;
    public boolean contains(Key key) {
        return get(key) != null;
    }
    public void delete(Key key) {
        put(key, null);
    }
}
```
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)` if and only if `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, File, URL, …

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;
    }
}
```

```
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;
        if (y.getClass() != this.getClass())
            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;
    }
}
```

Implementing equals for user-defined types

Seems easy, but requires some care.

```
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;
        if (y.getClass() != this.getClass())
            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;
    }
}
```

Equals design

"Standard" recipe for user-defined types.
- Optimization for reference equality.
- Check against null.
- Check that two objects are of the same type and 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.
- Only use necessary fields, e.g. a webpage is best defined by URL, not number of views.
- Make `compareTo()` consistent with `equals()`.
ST test client for traces

Build ST by associating value $i$ with $i$th string from standard input.

```java
public static void main(String[] args) {
    ST<String, Integer> st = new ST<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));
}
```

output

```
A 8
C 4
E 12
H 5
L 11
M 9
P 10
R 3
S 0
X 7
```

The order of output depends on the underlying data structure!

ST test client for analysis

**Frequency counter.** Read a sequence of strings from standard input and print out one that occurs with highest frequency.

```java
public class FrequencyCounter {
    public static void main(String[] args) {
        int minlen = Integer.parseInt(args[0]);
        ST<String, Integer> st = new ST<String, Integer>();
        while (!StdIn.isEmpty()) {
            String word = StdIn.readString();
            if (word.length() < minlen) 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));
    }
}
```

输出示例

```
tiny example: (60 words, 20 distinct)
real example: (135,635 words, 10,769 distinct)
real example: (21,191,455 words, 534,580 distinct)
```

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.

---

Elementary ST implementations: summary

<table>
<thead>
<tr>
<th>ST implementation</th>
<th>worst case</th>
<th>average case</th>
<th>ordered?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N / 2</td>
<td>no</td>
<td>equals()</td>
</tr>
</tbody>
</table>

---

Elementary ST implementations: summary

<table>
<thead>
<tr>
<th>ST implementation</th>
<th>worst case cost (after N inserts)</th>
<th>average case (after N random inserts)</th>
<th>ordered insertion?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N / 2</td>
<td>no</td>
<td>equals()</td>
</tr>
</tbody>
</table>

---

Challenge. Efficient implementations of both search and insert.

Binary search

**Data structure.** Maintain an ordered array of key-value pairs.

**Rank helper function.** How many keys < $k$?

---
**Binary search: Java implementation**

```java
public Value get(Key key)
{
    if (isEmpty()) return null;
    int i = rank(key);
    if (i < N && keys[i].compareTo(key) == 0) return vals[i];
    else return null;
}
```

```java
private int rank(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 mid;
    }
    return lo;
}
```

**Binary search: mathematical analysis**

**Proposition.** Binary search uses \( \sim \lg N \) compares to search any array of size \( N \).

**Pf.**

\[
T(N) = \text{number of compares to binary search in a sorted array of size } N.
\]

\[
\leq T(\lfloor N/2 \rfloor) + 1
\]

Recall lecture 2.

**Elementary ST implementations: frequency counter**

**Problem.** To insert, need to shift all greater keys over.

---

**Problem.** To insert, need to shift all greater keys over.
Elementary ST implementations: summary

<table>
<thead>
<tr>
<th>ST implementation</th>
<th>worst-case cost (after N inserts)</th>
<th>average case (after N random inserts)</th>
<th>ordered iteration?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>N</td>
<td>N / 2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>insert</td>
<td>N</td>
<td>N / 2</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>search hit</td>
<td>N</td>
<td>N / 2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>insert</td>
<td>N</td>
<td>N / 2</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N / 2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>binary search</td>
<td>log N</td>
<td>N / 2</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

Challenge. Efficient implementations of both search and insert.

Ordered symbol table API (Example Operations)

```
keys(09:15:00, 09:25:00)
size(09:15:00, 09:25:00) = 5
rank(09:10:25) = 7

09:00:00 Chicago
09:00:03 Phoenix
09:00:11 Houston
09:00:59 Chicago
09:01:10 Houston
09:10:11 Seattle
09:14:25 Phoenix
09:19:32 Chicago
09:19:46 Chicago
09:21:05 Chicago
09:22:43 Seattle
09:22:54 Seattle
09:25:52 Chicago
09:35:21 Chicago
09:36:14 Seattle
09:37:44 Phoenix
```

Examples of ordered symbol-table operations

```
get(09:00:13)  —  Chicago
ceiling(09:30:00)  —  Chicago
keys(09:15:00, 09:25:00)
size(09:15:00, 09:25:00)
```

Ordered symbol table API

```
public class ST<Key extends Comparable<Key>, Value>  
create an ordered symbol table
ST()
void put(Key key, Value val)  
Value get(Key key)
boolean contains(Key key)
boolean isEmpty()
int size()
Key min()
Key max()
Key floor(Key key)
Key ceiling(Key key)
int rank(Key key)
Key select(int k)
void deleteMin()
void deleteMax()
int size(Key lo, Key hi)
Iterable<Key> keys(Key lo, Key hi)
Iterable<Key> keys()
```
## Binary search: ordered symbol table operations summary

<table>
<thead>
<tr>
<th>Operation</th>
<th>Sequential Search Time</th>
<th>Binary Search Time</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Search</strong></td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td><strong>Insert</strong></td>
<td>$1$</td>
<td>$N$</td>
</tr>
<tr>
<td><strong>Min / Max</strong></td>
<td>$N$</td>
<td>$1$</td>
</tr>
<tr>
<td><strong>Floor / Ceiling</strong></td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td><strong>Rank</strong></td>
<td>$N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td><strong>Select</strong></td>
<td>$N$</td>
<td>$1$</td>
</tr>
<tr>
<td><strong>Ordered Iteration</strong></td>
<td>$N \log N$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

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

**The Problem: Insert Operation**