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>130.132.143.21</td>
</tr>
<tr>
<td><a href="http://www.harvard.edu">www.harvard.edu</a></td>
<td>128.103.060.55</td>
</tr>
<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
public boolean contains(Key key) {
  return get(key) != null;
}
```

- Can implement lazy version of delete().

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

Basic symbol table API

Associative array abstraction. Associate one value with each key.

```java
public class ST<Key, Value> {

  public ST() {  create a symbol table
  }

  void put(Key key, Value val) {
  put key-value pair into the table
  (remove key from table if value is null)
  value paired with key
  (null) if key is absent
  }

  Value get(Key key) {
  is there a value paired with key?
  }

  void delete(Key key) {
  remove key and its value from table
  is the table empty?
  number of key-value pairs in the table
  all the keys in the table
  }

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

  boolean isEmpty() {
  return size() == 0;
  }

  int size() {
  return keys().size();
  }

  Iterable<Key> keys() {
  return 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, ...
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.


Implementing equals for user-defined types

Seems easy.

```
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

Typically unsafe to use `equals()` with inheritance (would violate symmetry). Must be `Object`. Why? Experts still debate. Optimize for true object equality. Check for null. Objects must be in the same class (religion: `getClass()` vs. `instanceof`).

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) if and only if (x.compareTo(y) == 0)
```

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
- Apply rule recursively alternatively, use `Arrays.equals(a, b)` or `Arrays.deepEquals(a, b)`, but not `a.equals(b)`.
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));
}
```

<table>
<thead>
<tr>
<th>keys</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>S A R C H E X A M P L E</td>
<td>0 1 2 3 4 5 6 7 8 9 10 11 12</td>
</tr>
</tbody>
</table>

ST test client for analysis

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

```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
%
java FrequencyCounter 1 < tinyTale.txt
it 10
%
java FrequencyCounter 8 < tale.txt
business 122
%
java FrequencyCounter 10 < leipzig1M.txt
government 24763
```

Frequency counter implementation

```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));
    }
}
```

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 cost (after N inserts)</th>
<th>average-case cost (after N random inserts)</th>
<th>ordered iteration?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N / 2</td>
<td>N</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 ?

---

Trace of binary search for rank in an ordered array
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;
}

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.** \( T(N) \equiv \) number of compares to binary search in a sorted array of size \( N \).

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

Recall lecture 2.

Binary search: trace of standard indexing client

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

<table>
<thead>
<tr>
<th>keys[]</th>
<th>vals[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>S 0</td>
<td>S 1</td>
</tr>
<tr>
<td>E 1</td>
<td>E 2</td>
</tr>
<tr>
<td>A 2</td>
<td>A 3</td>
</tr>
<tr>
<td>R 3</td>
<td>R 4</td>
</tr>
<tr>
<td>C 4</td>
<td>C 5</td>
</tr>
<tr>
<td>H 5</td>
<td>H 6</td>
</tr>
<tr>
<td>E 6</td>
<td>E 7</td>
</tr>
<tr>
<td>A 8</td>
<td>A 9</td>
</tr>
<tr>
<td>M 9</td>
<td>M 10</td>
</tr>
<tr>
<td>P 10</td>
<td>P 11</td>
</tr>
<tr>
<td>L 11</td>
<td>L 12</td>
</tr>
<tr>
<td>E 12</td>
<td>E 13</td>
</tr>
</tbody>
</table>

Elementary ST implementations: frequency counter

**Costs for** java FrequencyCounter 8 < tale.txt **using** BinarySearchST

**Costs for** java FrequencyCounter 8 < tale.txt **using** SequentialSearchST
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>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log N</td>
<td>N</td>
<td>log N</td>
<td>N/2</td>
</tr>
</tbody>
</table>

Challenge. Efficient implementations of both search and insert.

Ordered symbol table API

Examples of ordered symbol-table operations

public class ST<Key extends Comparable<Key>, Value>

ST() create an ordered symbol table
void put(Key key, Value val) put key-value pair into the table (remove key from table if value is null)
Value get(Key key) value paired with key (null if key is absent)
void delete(Key key) remove key (and its value) from table
boolean contains(Key key) is there a value paired with key?
boolean isEmpty() is the table empty?
int size() number of key-value pairs
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
void deleteMin() delete smallest key
void deleteMax() delete largest key
int size(Key lo, Key hi) number of keys in [lo, hi]
Iterable<Key> keys(Key lo, Key hi) keys in [lo, hi], in sorted order
Iterable<Key> keys() all keys in the table, in sorted order
# 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>$I$</td>
<td>$N$</td>
</tr>
<tr>
<td>min / max</td>
<td>$N$</td>
<td>$I$</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>$I$</td>
</tr>
<tr>
<td>ordered iteration</td>
<td>$N \log N$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

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