Symbol Tables

- API
- Elementary implementations
- Ordered operations

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>128.103.040.55</td>
</tr>
<tr>
<td><a href="http://www.simpsons.com">www.simpsons.com</a></td>
<td>209.052.145.60</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 ST<Key, Value> {
    public boolean contains(Key key) {
        return get(key) != null;
    }
}
```

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

```java
class ST<Key, Value> {
    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> {
    create a symbol table
    put key-value pair into the table
    remove key from table if value is not null
    value paired with key
    (null) if key is absent
    remove key and its value from table
    is there a value paired with key?
    is the table empty?
    number of key-value pairs in the table
    all the keys in the table
    public ST() {
    }
    void put(Key key, Value val) {
        put(key, val);
    }
    Value get(Key key) {
        return a[key];
    }
    void delete(Key key) {
        remove(key);
    }
    boolean contains(Key key) {
        return a.containsKey(key);
    }
    boolean isEmpty() {
        return a.isEmpty();
    }
    int size() {
        return a.size();
    }
    Iterable<Key> keys() {
        return a.keySet();
    }
}
```

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, ...`
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`, `File`, `URL`, ...

User-defined implementations. Some care needed.

Implementing equals for user-defined types

Seems easy.

```java
public final 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;
    }
}
```

Typically unsafe to use `equals()` with inheritance (would violate symmetry)

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>S E A R C H E X A M P L E</th>
</tr>
</thead>
<tbody>
<tr>
<td>values</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 (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 / 2</td>
<td>no</td>
<td>equals()</td>
</tr>
</tbody>
</table>

---

Elementary ST implementations: summary

<table>
<thead>
<tr>
<th>ST implementation</th>
<th>search</th>
<th>insert</th>
<th>search hit</th>
<th>insert</th>
<th>search hit</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N / 2</td>
<td>N</td>
<td>no</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;
}
```

```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.**

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

Recall lecture 2.

Elementary ST implementations: frequency counter

Costs for java FrequencyCounter on file tale.txt using SequentialSearchST:

- 57,378 operations
- 2,246 compares

Costs for java FrequencyCounter on file tale.txt using BalancedSearchST:

- 57,378 operations
- 1,277 compares

Costs for java FrequencyCounter on file tale.txt using BinarySearchST:

- 57,378 operations
- 966 compares

Trace of standard indexing client

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

- Key values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, N
- Values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

Entries in gray did not move

Entries in black were inserted

Entries in red were moved to the right

Entries that were changed values

Elements in gray did not move

Elements in black were changed values

Elements in red were inserted
### 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/2</td>
<td>N</td>
<td>no</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

```
keys() — 09:00:00 Chicago
09:00:03 Phoenix
09:00:13 Houston
09:00:17 Las Vegas
09:00:59 Chicago
09:01:10 Houston
09:01:31 Chicago
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:

- `min()`: Return the key with the minimum value.
- `max()`: Return the key with the maximum value.
- `floor(Key key)`: Return the largest key less than or equal to `key`.
- `ceiling(Key key)`: Return the smallest key greater than or equal to `key`.
- `rank(Key key)`: Return the number of keys less than `key`.
- `select(int k)`: Return the key of rank `k`.
- `put(Key key, Value val)`: Put key-value pair into the table. (Remove key from table if value is null.)
- `get(Key key)`: Return the value paired with `key` (null if `key` is absent).
- `contains(Key key)`: Return true if `key` is in the symbol table.
- `isEmpty()`: Return true if the symbol table is empty.
- `size()`: Return the number of key-value pairs in the symbol table.
- `size(Key lo, Key hi)`: Return the number of keys in `[lo..hi]`.
- `keys(Key lo, Key hi)`: Return an iterator over all keys in `[lo..hi]`, in sorted order.
- `keys()`: Return an iterator over all keys in the table, in sorted order.

### Symbol Tables

- **API**
- **Elementary implementations**
- **Ordered operations**
### 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>$\lg N$</td>
</tr>
<tr>
<td>insert</td>
<td>$1$</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>$\lg N$</td>
</tr>
<tr>
<td>rank</td>
<td>$N$</td>
<td>$\lg N$</td>
</tr>
<tr>
<td>select</td>
<td>$N$</td>
<td>$1$</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.