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>URL</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>

Basic symbol table API

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

```java
public class ST<Key, Value>
{
    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 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?
    is the table empty?
    int size() number of key-value pairs in the table
    number of key-value pairs in the table
    Iterable<Key> keys() all the keys in the 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);
}
```

**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(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

Safer to use `equals()` with inheritance if fields in extending class contribute to `equals()`: the symmetry violated

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()`.

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.
- 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));
}
```

Output:
```
keys SEARCH EXAMPLE
values 0 1 2 3 4 5 6 7 8 9 10 11 12
```

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.

```
% 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</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.

---

**Elementary ST implementations: summary**

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

Grey data points are observed costs for i\(^{th}\) operation, reds are their averages.

---

**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.**

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

- left or right half $\leq T([N/2]) + 1$

Recall lecture 2.

### Binary search: trace of standard indexing client

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

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

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

### Elementary ST implementations: frequency counter

Costs for java FrequencyCounter 8 < tale.txt using BinarySearchST

<table>
<thead>
<tr>
<th>operations</th>
<th>compares</th>
</tr>
</thead>
<tbody>
<tr>
<td>14350</td>
<td>5737</td>
</tr>
<tr>
<td>2246</td>
<td>5737</td>
</tr>
</tbody>
</table>

Costs for java FrequencyCounter 8 < tale.txt using SequentialSearchST

<table>
<thead>
<tr>
<th>operations</th>
<th>compares</th>
</tr>
</thead>
<tbody>
<tr>
<td>14350</td>
<td>5737</td>
</tr>
<tr>
<td>2246</td>
<td>5737</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 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>no</td>
<td>compareTo()</td>
</tr>
<tr>
<td>search hit</td>
<td>N/2</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)  5
size(09:15:00, 09:25:00)  5
rank(09:15:25)  7
min()  09:00:00 Chicago
max()  09:25:52 Phoenix
floor(09:05:00)  09:00:11 Houston
ceiling(09:30:00)  09:30:11 Chicago
keys()  09:00:00 Chicago
keys(09:00:00)  Chicago
keys(09:00:11)  Houston
keys(09:00:13)  Houston
keys(09:00:59)  Chicago
keys(09:10:11)  Seattle
keys(09:10:25)  Seattle
keys(09:14:25)  Phoenix
keys(09:19:32)  Chicago
keys(09:19:46)  Chicago
keys(09:21:05)  Philadelphia
keys(09:22:43)  Seattle
keys(09:22:54)  Seattle
keys(09:25:52)  Chicago
keys(09:35:21)  Chicago
keys(09:36:14)  Seattle
keys(09:37:44)  Phoenix
```

Examples of ordered symbol-table operations

**Orders of elements**

- **get()**
  - get(09:00:13) returns Chicago
  - get(09:00:13) returns Houston
- **ceiling()**
  - ceiling(09:30:00) returns Chicago
  - ceiling(09:30:00) returns Houston
- **floor()**
  - floor(09:05:00) returns Houston
  - floor(09:05:00) returns Houston
- **select()**
  - select(7) returns 09:10:25 Seattle
- **size()**
  - size(09:15:00, 09:25:00) returns 5
  - size(09:15:00, 09:25:00) returns 5
- **rank()**
  - rank(09:10:25) returns 7
  - rank(09:10:25) returns 7

---

Ordered symbol table API

```
public class ST<Key extends Comparable<Key>, Value> {  
  create an ordered symbol table
  void put(Key key, Value val) { ... }  
  Value get(Key key) { ... }  
  void delete(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</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>

The Problem:
Insert Operation

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