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.

Example: 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 webpages</td>
<td>keyword</td>
<td>list of page names</td>
</tr>
<tr>
<td>compiler</td>
<td>find properties of</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</td>
<td>URL</td>
<td>IP address</td>
</tr>
<tr>
<td>reverse DNS</td>
<td>find URL given IP</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

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) { value paired with key } 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 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`, ...

Built-in to Java (stay tuned)
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 Data implements Comparable<Data> {
    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
Safer to use `equals()` with inheritance if fields in extending class contribute to `equals()` the symmetry violation

Implementing equals for user-defined types

Seems easy, but requires some care.

```java
public final class Data implements Comparable<Data> {
    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 = (Data) y;
        if (this.day   != that.day  ) return false;
        if (this.month != that.month) return false;
        if (this.year  != that.year ) return false;
        return true;
    }
}
```

Best practices.
- No need to use calculated fields that depend on other fields.
- 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
- `a.equals(b)` if and only if `(a.compareTo(b) == 0)`

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));
}
```

The order of output depends on the underlying data structure!

<table>
<thead>
<tr>
<th>keys</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEARCH</td>
<td>1</td>
</tr>
<tr>
<td>EXAMPLE</td>
<td>12</td>
</tr>
<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>S</td>
<td>8</td>
</tr>
<tr>
<td>E</td>
<td>9</td>
</tr>
<tr>
<td>A</td>
<td>10</td>
</tr>
<tr>
<td>R</td>
<td>11</td>
</tr>
<tr>
<td>C</td>
<td>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 < tales.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.

Challenge. Efficient implementations of both search and insert.

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/2 )</td>
<td>( N )</td>
<td>no</td>
</tr>
</tbody>
</table>

Binary search

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

Rank helper function. How many keys < \( k \)?

Challenge. Efficient implementations of both search and insert.
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.

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

Costs for java FrequencyCounter 8 < tale.txt using SequentialSearchST
```

Elementary ST implementations: frequency counter
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</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>
</tbody>
</table>

Sequential search (unordered list)

- $O(N)$
- $O(N/2)$

Binary search (ordered array)

- $O(log N)$
- $O(N)$

**Challenge.** Efficient implementations of both search and insert.

### Ordered symbol table API

#### Example Operations

- **min()**
  - 09:00:00 Chicago
  - 09:00:03 Phoenix

- **get(09:00:13)**
  - 09:00:19 Chicago

- **floor(09:05:00)**
  - 09:03:33 Chicago

- **select(7)**
  - 09:10:23 Seattle

- **keys(09:15:00, 09:25:00)**
  - 09:19:32 Chicago

- **ceilings(09:30:00)**
  - 09:37:12 Chicago

- **max()**
  - 09:37:44 Phoenix

**Examples of ordered symbol-table operations**

- $size(09:11:00, 09:25:00)$ is 5
- $rank(09:10:25)$ is 7

### API

- **create an ordered symbol table**
- **put(key key, Value val)**
  - put key-value pair into the table
  - value paired with key
  - (null if key is absent)
- **get(Key key)**
  - is there a value paired with key?
- **delete(Key key)**
  - is the table empty?
- **contains(Key key)**
  - number of key-value pairs
- **isEmpty()**
  - smallest key
- **min()**
  - largest key
- **max()**
  - largest key less than or equal to key
- **floor(Key key)**
  - smallest key greater than or equal to key
- **ceil(Key key)**
  - number of keys less than key
- **select(int k)**
  - key of rank k
- **deleteMin()**
  - delete smallest key
- **deleteMax()**
  - delete largest key
- **size(Key lo, Key hi)**
  - number of keys in [lo, hi]
- **keys(Key lo, Key hi)**
  - keys in [lo, hi], in sorted order
- **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>$N$</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>
<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.