3.1 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 domain name with specified IP address.
- Given domain name, find corresponding IP address.

<table>
<thead>
<tr>
<th>domain name</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</td>
<td>domain name</td>
<td>IP address</td>
</tr>
<tr>
<td>reverse DNS</td>
<td>find domain name</td>
<td>IP address</td>
<td>domain name</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>
Symbol tables: context

Also known as: maps, dictionaries, associative arrays.

Generalizes arrays. Keys need not be between 0 and N – 1.

Language support.
- External libraries: C, VisualBasic, Standard ML, bash, ...
- Built-in libraries: Java, C#, C++, Scala, ...
- Built-in to language: Awk, Perl, PHP, Tcl, JavaScript, Python, Ruby, Lua.

Basic symbol table API

Associative array abstraction. Associate one value with each key.

Java allows null value

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: Integer, Double, String, 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().
- Can implement lazy version of delete().
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 \( y.equals(x) \).
- 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, java.io.File, ...

User-defined implementations. Some care needed.

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 == null) return false;
        if (y.getClass() != getClass()) {
            return false;
        }
        if (this.day != that.day) return false;
        if (this.month != that.month) return false;
        return true;
    }
}
```

```
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;
}
```

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.
- Make compareTo() consistent with equals().

typically unsafe to use equals() with inheritance
(would violate symmetry)

objects must be in the same class
(religion: getClass() vs. instanceof)

cast is guaranteed to succeed
check that all significant fields are the same

e.g., cached Manhattan distance

but use Double.compare() with double
(or otherwise deal with -0.0 and NaN)
apply rule recursively

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</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>X</td>
<td>7</td>
</tr>
<tr>
<td>A</td>
<td>8</td>
</tr>
<tr>
<td>M</td>
<td>9</td>
</tr>
<tr>
<td>P</td>
<td>10</td>
</tr>
<tr>
<td>L</td>
<td>11</td>
</tr>
<tr>
<td>S</td>
<td>12</td>
</tr>
</tbody>
</table>

output:

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

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.containsKey(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));
    }
}
```

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
% java FrequencyCounter 1 < tinyTale.txt
it 10
```

```java
% java FrequencyCounter 8 < tale.txt
business 122
```

```java
% java FrequencyCounter 10 < leipzig3M.txt
government 24763
```

3.1 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.

### Binary search in an ordered array

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

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

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

### Elementary ST implementations: summary

<table>
<thead>
<tr>
<th>ST implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N/2</td>
<td>equals()</td>
</tr>
<tr>
<td>insert search</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>search hit</td>
<td>N</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td>N</td>
<td>N</td>
<td>equals()</td>
</tr>
</tbody>
</table>

### 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;
}
```
Problem. To insert, need to shift all greater keys over.

Binary search: trace of standard indexing client

<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>A</td>
<td>8</td>
</tr>
<tr>
<td>M</td>
<td>9</td>
</tr>
<tr>
<td>P</td>
<td>10</td>
</tr>
<tr>
<td>L</td>
<td>11</td>
</tr>
<tr>
<td>E</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>keys[]</th>
<th>vals[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 2 3 4 5 6 7 8 9</td>
<td>N 0 1 2 3 4 5 6 7 8 9</td>
</tr>
</tbody>
</table>

| 0 1 2 3 4 5 6 7 8 9 | 0 1 2 3 4 5 6 7 8 9 |

| 0 1 2 3 4 5 6 7 8 9 | 0 1 2 3 4 5 6 7 8 9 |

Challenge. Efficient implementations of both search and insert.

Elementary ST implementations: summary

<table>
<thead>
<tr>
<th>ST implementation</th>
<th>guarantee</th>
<th>average case</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>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log N</td>
<td>N</td>
<td>log N</td>
</tr>
</tbody>
</table>

Examples of ordered symbol-table operations

key: 09:00:00 value: Chicago
key: 09:00:03 value: Phoenix
key: 09:00:13 value: Houston
key: 09:00:59 value: Chicago
key: 09:10:25 value: Seattle
key: 09:14:25 value: Phoenix
key: 09:19:32 value: Chicago
key: 09:19:46 value: Chicago
key: 09:21:05 value: Chicago
key: 09:22:43 value: Seattle
key: 09:22:54 value: Seattle
key: 09:25:52 value: Chicago
key: 09:35:21 value: Chicago
key: 09:36:14 value: Seattle
key: 09:37:44 value: Phoenix

Examples of ordered symbol table API

keys: 09:00:00, 09:25:00
values: Chicago, Phoenix

select(7) 09:10:25 Seattle
floor(09:05:00) 09:03:13 Chicago
get(09:00:13) 09:00:59 Chicago
get(09:00:03) 09:00:03 Phoenix
get(09:00:13) 09:00:13 Houston
get(09:10:11) 09:10:11 Seattle

3.1 SYMBOL TABLES

- API
- elementary implementations
- ordered operations

key: 09:00:00 value: Chicago
key: 09:00:03 value: Phoenix
key: 09:00:13 value: Houston
key: 09:00:59 value: Chicago
key: 09:10:25 value: Seattle
key: 09:14:25 value: Phoenix
key: 09:19:32 value: Chicago
key: 09:19:46 value: Chicago
key: 09:21:05 value: Chicago
key: 09:22:43 value: Seattle
key: 09:22:54 value: Seattle
key: 09:25:52 value: Chicago
key: 09:35:21 value: Chicago
key: 09:36:14 value: Seattle
key: 09:37:44 value: Phoenix

size(09:15:00, 09:25:00) is 5
rank(09:10:25) is 7
Ordered symbol table API

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

...)

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 between lo and hi
Iterable<Key> keys() all keys, in sorted order
Iterable<Key> keys(Key lo, Key hi) keys between lo and hi, 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 / delete</td>
<td>N</td>
<td><strong>N</strong></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><strong>N log N</strong></td>
<td>N</td>
</tr>
</tbody>
</table>

order of growth of the running time for ordered symbol table operations