3.1 Symbol Tables

- **API**
  - sequential search
  - binary search
  - ordered operations

### Symbol Table API

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

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

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 get(Key key) value paired with key
    (null if key is absent)
void delete(Key key) remove key (and its value) from table
    is there a value paired with key?
boolean contains(Key key) is the table empty?
    number of key-value pairs in the table
boolean isEmpty() all the keys in the table
int size() API for a generic basic symbol table
Iterable<Key> keys()
```

### 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>

### API

- **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>
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()`.

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

```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()`. (built-in to Java)
- Assume keys are any generic type, use `equals()` to test equality.
- Assume keys are any generic type, use `equals()` to test equality and `hashCode()` to scramble key.

Best practices. Use immutable types for symbol table keys.

- Immutable in Java: `String`, `Integer`, `Double`, `File`, ...
- Mutable in Java: `Date`, `StringBuilder`, `Url`, ...

Implementing `equals()` for user-defined types

Seems easy

```java
public class Record {
    private final String name;
    private final long val;
    ...
    public boolean equals(Record y) {
        Record that = y;
        return (this.val == that.val) && (this.name.equals(that.name));
    }
}
```

Seems easy, but requires some care.

```java
public final class Record {
    private final String name;
    private final long val;
    ...
    public boolean equals(Object y) {
        if (y == this) return true;
        if (y == null) return false;
        if (y.getClass() != this.getClass()) return false;
        Record that = (Record) y;
        return (this.val == that.val) && (this.name.equals(that.name));
    }
}
```
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));
    }
}
```

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
```
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</th>
<th>average case</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N/2</td>
<td>N</td>
<td>no equals()</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 recurrence**

- **Binary search recurrence.** \( T(N) \leq T(N/2) + 1 \) for \( N > 1 \), with \( T(1) = 1 \).

**Proposition.** If \( N \) is a power of 2, then \( T(N) \leq \lg N + 1 \).

**Pf.**

\[
T(N) \\
\leq T(N/2) + 1 \\
\leq T(N/4) + 1 + 1 \\
\leq T(N/8) + 1 + 1 + 1 \\
\vdots \\
\leq T(N/N) + 1 + 1 + \ldots + 1 \\
= \lg N + 1
\]

Given

apply recurrence to first term

apply recurrence to first term

stop applying, \( T(1) = 1 \)

**Binary search: mathematical analysis**

**Proposition.** Binary search uses \( \lg N \) compares to search any array of size \( N \).

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

\[
T(N) = T(N/2) + 1 \\
\text{left or right half}
\]

**Binary search recurrence.** \( T(N) \leq T(N/2) + 1 \) for \( N > 1 \), with \( T(1) = 1 \).

- Not quite right for odd \( N \).
- Same recurrence holds for many algorithms.

**Solution.** \( T(N) \sim \lg N \).

- For simplicity, we’ll prove when \( N \) is a power of 2.
- True for all \( N \). [see COS 340]

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

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

<table>
<thead>
<tr>
<th>key</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>9</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
<td>11</td>
<td>9</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>4</td>
<td>12</td>
<td>5</td>
<td>11</td>
<td>9</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>11</td>
<td>9</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

entries in gray did not move

entries in black moved to the right

entries in red were inserted

entries in cyan changed values
**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 (unordered list)</td>
<td>N</td>
<td>N</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log N</td>
<td>N</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

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

**Ordered symbol table API**

```java
public class ST<Key extends Comparable<Key>, Value> {
    // API
    void put(Key key, Value val) // put key-value pair into the table
    Value get(Key key) // remove key from table (if value is null) value paired with key
    boolean contains(Key key) // is there a value paired with key? number of key-value pairs
    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 delete(Key key) // delete key-value pair
    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
}
```

**Examples of ordered symbol-table operations**

- `min()`: 09:00:00 Chicago
- `09:00:03 Phoenix
- `09:00:13 Houston
- `get(09:00:13)`: 09:00:13 Houston
- `floor(09:00:13)`: 09:00:10 Seattle
- `select(7)`: 09:10:11 Seattle
- `ceiling(09:30:00)`: 09:35:21 Chicago
- `max()`: 09:37:44 Phoenix

**Costs for Java FrequencyCounter: B < tale.txt using OrderedArrayST**

```
- 5001
- 464
```

**API**

- `sequential search`
- `binary search`
- `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>

Worst-case running time of ordered symbol table operations