### 3.1 Symbol Tables

- **API**
- **elementary implementations**
- **ordered operations**

#### Data structures

> “Smart data structures and dumb code works a lot better than the other way around.”  
— Eric S. Raymond

---

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

- **public class ST<Key, Value>**
  - create an empty symbol table
  - void put(Key key, Value val) put key-value pair into the table
  - Value get(Key key) value paired with key
  - boolean contains(Key key) is there a value paired with key?
  - Iterable<Key> keys() all the keys in the table
  - void delete(Key key) remove key (and its value) from table
  - boolean isEmpty() is the table empty?
  - int size() number of key-value pairs in the table

Conventions

- Values are not null. → java.util.Map allows null values
- 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: Integer, Double, String, java.io.File, ...
- Mutable in Java: StringBuilder, java.net.URL, arrays, ...

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;
    }
}
```

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, java.io.File, ...
User-defined implementations. Some care needed.

Implementing equals for user-defined types

Seems easy, but requires some care.

```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 == 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;
    }
}
```
Equals design

"Standard" recipe for user-defined types.
• Optimization for reference equality.
• Check against null.
• Check that two objects are of the same type; 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

but use Double.compare() with double
(to deal with -0.0 and NaN)
apply rule recursively
can use Arrays.deepEquals(a, b)
but not a.equals(b)

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)

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 3 < tinyTale.txt
the 10
% java FrequencyCounter 8 < tale.txt
business 122
% java FrequencyCounter 10 < leipzig2M.txt
government 24763

tiny example (60 words, 20 distinct)
real example (135,635 words, 10,769 distinct)
real example (21,191,455 words, 534,580 distinct)

Frequency counter implementation

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));
    }
}

3.1 SYMBOL TABLES

‣ API
‣ elementary implementations
‣ ordered operations

http://algs4.cs.princeton.edu

Robert Sedgewick | Kevin Wayne
Algorithms
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.

get("A")

```
0 1 2 3 4 5 6 7 8 9
A C E H L M P R S X
```

put("M", 9)

```
0 1 2 3 4 5 6 7 8 9
A C E H L M P R S X
```

Binary search in an ordered array

Data structure. Maintain parallel arrays for keys and values, sorted by keys.

Search. Use binary search to find key.

Proposition. At most –lg

Search. Use binary search to find key.

```java
public Value get(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 vals[mid];
    }
    return null; // no matching key
}
```
Elementary symbol tables: quiz 1

Implementing binary search was

A. Easier than I thought.
B. About what I expected.
C. Harder than I thought.
D. Much harder than I thought.
E. I don’t know.

Problem. Given an array with all 0s in the beginning and all 1s at the end, find the index in the array where the 1s begin.

input
0 0 0 0 0 ... 0 0 0 0 0 1 1 1 ... 1 1 1

Variant 1. You are given the length of the array.
Variant 2. You are not given the length of the array.

Elementary ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>search</td>
<td>search hit</td>
<td>insert</td>
</tr>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log N</td>
<td>log N</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>N</td>
<td>N</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

Challenge. Efficient implementations of both search and insert.
### 3.1 SYMBOL TABLES

- **API**
- **elementary implementations**
- **ordered operations**

#### Ordered symbol table API

```java
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
    // ...
}
```

#### Examples of ordered symbol table API

<table>
<thead>
<tr>
<th>keys</th>
<th>values</th>
</tr>
</thead>
<tbody>
<tr>
<td>min()</td>
<td>09:00:00</td>
</tr>
<tr>
<td></td>
<td>09:00:03</td>
</tr>
<tr>
<td></td>
<td>09:00:13</td>
</tr>
<tr>
<td>get(09:00:13)</td>
<td>09:00:59</td>
</tr>
<tr>
<td></td>
<td>09:01:10</td>
</tr>
<tr>
<td>floor(09:05:00)</td>
<td>09:03:13</td>
</tr>
<tr>
<td></td>
<td>09:10:11</td>
</tr>
<tr>
<td>select(7)</td>
<td>09:10:25</td>
</tr>
<tr>
<td></td>
<td>09:14:25</td>
</tr>
<tr>
<td></td>
<td>09:19:32</td>
</tr>
<tr>
<td></td>
<td>09:19:46</td>
</tr>
<tr>
<td>keys(09:15:00, 09:25:00)</td>
<td>09:21:05</td>
</tr>
<tr>
<td></td>
<td>09:22:43</td>
</tr>
<tr>
<td></td>
<td>09:22:54</td>
</tr>
<tr>
<td></td>
<td>09:25:52</td>
</tr>
<tr>
<td>ceiling(09:30:00)</td>
<td>09:35:21</td>
</tr>
<tr>
<td></td>
<td>09:36:14</td>
</tr>
<tr>
<td>max()</td>
<td>09:37:44</td>
</tr>
</tbody>
</table>

- `size(09:15:00, 09:25:00)` is 5
- `rank(09:10:25)` is 7

#### Problem

**Rank in a Sorted Array**

Given a sorted array of $N$ distinct keys, find the number of keys strictly less than a given query key.
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>
</tbody>
</table>

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

3.2 Binary Search Trees

- BSTs
- ordered operations
- iteration
- deletion (see book)

Binary search trees

Definition. A BST is a binary tree in symmetric order.

A binary tree is either:
- Empty.
- Two disjoint binary trees (left and right).

Symmetric order. Each node has a key, and every node’s key is:
- Larger than all keys in its left subtree.
- Smaller than all keys in its right subtree.
Binary search tree demo

**Search.** If less, go left; if greater, go right; if equal, search hit.

successful search for H

![Binary search tree demo](image)

**Insert.** If less, go left; if greater, go right; if null, insert.

insert G

![Binary search tree demo](image)

**BST representation in Java**

**Java definition.** A BST is a reference to a root `Node`.

A `Node` is composed of four fields:
- A `Key` and a `Value`.
- A reference to the left and right subtree.

```
private class Node {
    private Key key;
    private Value val;
    private Node left, right;
    public Node(Key key, Value val) {
        this.key = key;
        this.val = val;
    }
}
```

Key and Value are generic types; Key is Comparable

**BST implementation (skeleton)**

```
public class BST<Key extends Comparable<Key>, Value> {
    private Node root;

    private class Node {
        /* see previous slide */
    }

    public void put(Key key, Value val) {
        /* see next slides */
    }

    public Value get(Key key) {
        /* see next slides */
    }

    public Iterable<Key> iterator() {
        /* see slides in next section */
    }

    public void delete(Key key) {
        /* see textbook */
    }
}
```
**BST search: Java implementation**

**Get.** Return value corresponding to given key, or null if no such key.

```java
public Value get(Key key) {
    Node x = root;
    while (x != null) {
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else if (cmp == 0) return x.val;
    }
    return null;
}
```

**Cost.** Number of compares = 1 + depth of node.

---

**BST insert: Java implementation**

**Put.** Associate value with key.

```java
public void put(Key key, Value val) {
    root = put(root, key, val);
}
```

```java
private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val);
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = put(x.left, key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else if (cmp == 0) x.val = val;
    return x;
}
```

**Cost.** Number of compares = 1 + depth of node.

---

**BST insert**

**Put.** Associate value with key.

Search for key, then two cases:
- Key in tree ⇒ reset value.
- Key not in tree ⇒ add new node.

---

**Tree shape**

- Many BSTs correspond to same set of keys.
- Number of compares for search/insert = 1 + depth of node.

---

**Bottom line.** Tree shape depends on order of insertion.
BST insertion: random order visualization

Ex. Insert keys in random order.

\[
N = 255 \\
\text{max} = 16 \\
\text{avg} = 9.1 \\
\text{opt} = 7.0
\]

Binary search trees: quiz 1

Given \( N \) distinct keys, what is the name of this sorting algorithm?

1. Shuffle the keys.
2. Insert the keys into a BST, one at a time.
3. Do an inorder traversal of the BST.

A. Insertion sort.
B. Mergesort.
C. Quicksort.
D. None of the above.
E. I don't know.

Correspondence between BSTs and quicksort partitioning

Correspondence is 1–1 if array has no duplicate keys.

BSTs: mathematical analysis

Proposition. If \( N \) distinct keys are inserted into a BST in random order, the expected number of compares for a search/insert is \( \sim 2 \ln N \).

Pf. 1–1 correspondence with quicksort partitioning.

Proposition. [Reed, 2003] If \( N \) distinct keys are inserted into a BST in random order, the expected height is \( \sim 4.311 \ln N \).

But... Worst-case height is \( N - 1 \).
[ exponentially small chance when keys are inserted in random order ]
### Binary search trees: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Operations on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (unordered list)</td>
<td>N</td>
<td>N</td>
<td>equals()</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>log N</td>
<td>N</td>
<td>compareTo()</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>log N</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>

Why not shuffle to ensure a (probabilistic) guarantee of log N?

### 3.2 Binary Search Trees

- **BSTs**
- **iteration**
- **ordered operations**
- **deletion**

#### Inorder traversal

**In what order does the traverse(root) code print out the keys in the BST?**

```java
private void traverse(Node x) {
    if (x == null) return;
    traverse(x.left);
    StdOut.println(x.key);
    traverse(x.right);
}
```

**A.** A C E H M R S X  
**B.** A C E R H M X S  
**C.** S E A C R H M X  
**D.** C A M H R E X S  
**E.** I don't know.

**Output:** A C E H M R S X
Inorder traversal

- Traverse left subtree.
- Enqueue key.
- Traverse right subtree.

**Property.** Inorder traversal of a BST yields keys in ascending order.

```java
public Iterable<Key> keys()
{
    Queue<Key> q = new Queue<Key>();
    inorder(root, q);
    return q;
}

private void inorder(Node x, Queue<Key> q)
{
    if (x == null) return;
    inorder(x.left, q);
    q.enqueue(x.key);
    inorder(x.right, q);
}
```

### LEVEL-ORDER TRAVERSAL

**Level-order traversal of a binary tree.**

- Process root.
- Process children of root, from left to right.
- Process grandchildren of root, from left to right.
- ...

```
level order traversal: S E T A R C H M
```

**Q1.** Given binary tree, how to compute level-order traversal?

```java
queue.enqueue(root);
while (!queue.isEmpty())
{
    Node x = queue.dequeue();
    if (x == null) continue;
    StdOut.println(x.item);
    queue.enqueue(x.left);
    queue.enqueue(x.right);
}
```

```
level order traversal: S E T A R C H M
```

**Q2.** Given level-order traversal of a BST, how to (uniquely) reconstruct BST?

**Ex.** `S E T A R C H M`
3.2 Binary Search Trees

- BSTs
- iteration
- ordered operations
- deletion

Minimum and maximum

Minimum. Smallest key in BST.
Maximum. Largest key in BST.

Q. How to find the min / max?

Floor and ceiling

Floor. Largest key in BST ≤ query key.
Ceiling. Smallest key in BST ≥ query key.

Q. How to find the floor / ceiling?

Computing the floor

Floor. Largest key in BST ≤ k?

Case 1. [ key in node x = k ]
The floor of k is k.

Case 2. [ key in node x > k ]
The floor of k is in the left subtree of x.

Case 3. [ key in node x < k ]
The floor of k can't be in left subtree of x: it is either in the right subtree of x or it is the key in node x.
Computing the floor

```java
public Key floor(Key key) {
    return floor(root, key);
}

private Key floor(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp == 0) return x;
    if (cmp < 0) return floor(x.left, key);
    Key t = floor(x.right, key);
    if (t != null) return t;
    return x.key;
}
```

Rank and select

**Q.** How to implement rank() and select() efficiently for BSTs?

**A.** In each node, store the number of nodes in its subtree.

Computing the rank

**Rank.** How many keys in BST < k?

**Case 1.** \( k < \text{key in node} \)
- No key in right subtree < k;
- some keys in left subtree < k.

**Case 2.** \( k > \text{key in node} \)
- All keys in left subtree < k;
- the key in the node is < k;
- some keys in right subtree may be < k.

**Case 3.** \( k = \text{key in node} \)
- All keys in left subtree < k;
- no key in right subtree < k.
Rank

**Rank.** How many keys in BST < k?

Easy recursive algorithm (3 cases!)

```java
public int rank(Key key) {
    return rank(key, root);
}

private int rank(Key key, Node x) {
    if (x == null) return 0;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) return rank(key, x.left);
    else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
    else if (cmp == 0) return size(x.left);
}
```

**ST implementations: summary**

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered ops?</th>
<th>key interface</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>search hit</td>
<td>insert</td>
</tr>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>log N</td>
<td>N</td>
<td>log N</td>
<td>N</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>log N</td>
<td>log N</td>
</tr>
<tr>
<td>red-black BST</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>

Next lecture. Guarantee logarithmic performance for all operations.