3.1 SYMBOL TABLES

- API
- elementary implementations
- ordered operations

---

"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.

```
is_awesome = ("Python": True, "Java": False)
print is_awesome["Python"]
```

Legal Python code

Basic symbol table API

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

```
public class ST<Key, Value> {
    private Map<Key, Value> st;

    ST() {
        create an empty symbol table
    }

    void put(Key key, Value val) {
        a[key] = val;
    }

    Value get(Key key) {
        value paired with key
        a[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.
- Method get() returns null if key not present.
- Method put() overwrites old value with new value.

Easy to implement contains().

```
public boolean contains(Key key) {
    return get(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 (next Wednesday).

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, ...

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.

```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;
    }
}
```

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 == 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;
    }
}
```
**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

**Useful for assignment**

**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)

---

**Frequency counter implementation**

```java
public class FrequencyCounter {
    public static void main(String[] args) {
        String word, key;
        while (!StdIn.isEmpty()) {
            word = StdIn.readString();
            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));
    }
}
```

---

**3.1 Symbol Tables**

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

**Search.** Use binary search to find key.

**Proposition.** At most \(-\log_2 N\) compares to search a sorted array of length \(N\).

---

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

```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.

Binary search: insert

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

Insert. Use binary search to find place to insert; shift all larger keys over.

Proposition. Takes linear time in the worst case.

```java
put("P", 10)
```

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></td>
<td>search</td>
<td>insert</td>
<td>search hit</td>
</tr>
<tr>
<td>sequential search</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
</tr>
<tr>
<td>unordered array or list</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$\log N$</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

```
public class ST<Key extends Comparable<Key>, Value> {
    Key min() { return smallest key; }
    Key max() { return largest key; }
    Key floor(Key key) { return largest key less than or equal to key; }
    Key ceiling(Key key) { return smallest key greater than or equal to key; }
    int rank(Key key) { return number of keys less than key; }
    Key select(int k) { return 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 Chicago</td>
</tr>
<tr>
<td>09:00:03 Phoenix</td>
<td></td>
</tr>
<tr>
<td>09:00:13 Houston</td>
<td></td>
</tr>
<tr>
<td>get(09:00:13)</td>
<td>09:00:59 Chicago</td>
</tr>
<tr>
<td>09:01:10 Houston</td>
<td></td>
</tr>
<tr>
<td>floor(09:05:00)</td>
<td>09:03:13 Chicago</td>
</tr>
<tr>
<td>09:10:11 Seattle</td>
<td></td>
</tr>
<tr>
<td>select(7)</td>
<td>09:10:25 Seattle</td>
</tr>
<tr>
<td>09:14:25 Phoenix</td>
<td></td>
</tr>
<tr>
<td>09:19:32 Chicago</td>
<td></td>
</tr>
<tr>
<td>09:19:46 Chicago</td>
<td></td>
</tr>
<tr>
<td>keys(09:15:00, 09:25:00)</td>
<td>09:21:05 Chicago</td>
</tr>
<tr>
<td>09:22:43 Seattle</td>
<td></td>
</tr>
<tr>
<td>09:22:54 Seattle</td>
<td></td>
</tr>
<tr>
<td>09:25:52 Chicago</td>
<td></td>
</tr>
<tr>
<td>ceiling(09:30:00)</td>
<td>09:35:21 Chicago</td>
</tr>
<tr>
<td>09:36:14 Seattle</td>
<td></td>
</tr>
<tr>
<td>max()</td>
<td>09:37:44 Phoenix</td>
</tr>
<tr>
<td>size(09:15:00, 09:25:00)</td>
<td>is 5</td>
</tr>
<tr>
<td>rank(09:10:25)</td>
<td>is 7</td>
</tr>
</tbody>
</table>
```

Rank in a sorted array

Problem. Given a sorted array of \( N \) distinct keys, find the number of keys strictly less than a given query key.

```
public Value get(Key key) public 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 vals[mid]; mid
    }
    return null; lo
}
```
### 3.2 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).

**Search tree.** 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 = Binary (search tree) = a search tree that's binary
also (Binary search) tree = a tree that supports binary search

Q. What are the differences between a heap and a binary search tree?
**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 {
        // ... (implementation details)
    }

    public void put(Key key, Value val) {
        // ... (implementation details)
    }

    public Value get(Key key) {
        // ... (implementation details)
    }

    public void delete(Key key) {
        // ... (implementation details)
    }

    public Iterable<Key> iterator() {
        // ... (implementation details)
    }
}
```

**BST Search**

**Search (get).**
Repeat:
- if less, _____
- if greater, _____
- if equal, _____
- if _____, search miss

```
only keys are shown
```

**BST Search**

**Search (get).**
Repeat:
- if less, go left;
- if greater, go right;
- if equal, return value (search hit)
- if null, return null (search miss)

```
only keys are shown
```
**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 put: non-recursive implementation**

**Repeat:**
- if less, ___
- if greater, ___
- if equal, ___
- if null, ___

**Put.** Associate value with key.

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

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;
}
```

**Warning:** concise but tricky code; read carefully!

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

Q. Draw the tree when the following keys are inserted: A, L, O, E, P, I, G, S

Q. Draw the tree when the following keys are inserted: A, E, G, I, L, O, P, S

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. \( \sim 2 \ln N \).

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></td>
<td>search</td>
<td>insert</td>
<td>search hit</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>( N )</td>
<td>( \log N )</td>
</tr>
<tr>
<td>BST</td>
<td>( N )</td>
<td>( N )</td>
<td>( \log N )</td>
</tr>
</tbody>
</table>

Why not shuffle to ensure a (probabilistic) guarantee of \( \log N \)?
### 3.2 Binary Search Trees

#### Inorder traversal

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

```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);
}
```

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

---

**Binary search trees: inorder traversal**

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

- 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. None of the above.

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

---

**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

Remark. 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.

But... Worst-case height is $N - 1$.

[when client provides keys, they may *not* be in random order, and we have no control over probability of worst case]

Binary search trees: preorder 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;
  StdOut.println(x.key);
  traverse(x.left);
  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 S X
E. None of the above.

Binary search trees: postorder 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);
  traverse(x.right);
  StdOut.println(x.key);
}
```

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 S X
E. None of the above.
Level-order traversal of a binary tree

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

```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);
}
```

Minimum and maximum

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

Q. How to find the min / max?

3.2 Binary Search Trees

**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 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.

---

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.

---

BST implementation: subtree counts

```java
private class Node {
    private Key key;
    private Value val;
    private Node left;
    private Node right;
    private int count;
}
```

```java
private int size(Node x) {
    if (x == null) return 0;
    return x.count;
}
```

```java
private Node put(Node x, Key key, Value val) {
    if (x == null) return new Node(key, val, 1);
    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;
    x.count = 1 + size(x.left) + size(x.right);
    return x;
}
```

```java
public int size() {
    return size(root);
}
```

---

Finding floor(S)

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

```java
int cmp = key.compareTo(x.key);
if (cmp == 0) return x;
if (cmp < 0) return floor(x.left, key);
return floor(x.right, key);
```
Computing the rank

**Rank.** How many keys in BST < \( k \)?

**Case 1.** \( k < \text{ key in node } \)
- Keys in left subtree? \( \text{count} \)
- Key in node? \( 0 \)
- Keys in right subtree? \( 0 \)

**Case 2.** \( k > \text{ key in node } \)
- Keys in left subtree? \( \text{all} \)
- Key in node. \( 1 \)
- Keys in right subtree? \( \text{count} \)

**Case 3.** \( k = \text{ key in node } \)
- Keys in left subtree? \( \text{count} \)
- Key in node. \( 0 \)
- Keys in right subtree? \( 0 \)

**BST: ordered symbol table operations summary**

<table>
<thead>
<tr>
<th></th>
<th>sequential search</th>
<th>binary search</th>
<th>BST</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>( N )</td>
<td>( \log N )</td>
<td>( h )</td>
</tr>
<tr>
<td>insert</td>
<td>( N )</td>
<td>( N )</td>
<td>( h )</td>
</tr>
<tr>
<td>min / max</td>
<td>( N )</td>
<td>( 1 )</td>
<td>( h )</td>
</tr>
<tr>
<td>floor / ceiling</td>
<td>( N )</td>
<td>( \log N )</td>
<td>( h )</td>
</tr>
<tr>
<td>rank</td>
<td>( N )</td>
<td>( \log N )</td>
<td>( h )</td>
</tr>
<tr>
<td>select</td>
<td>( N )</td>
<td>( 1 )</td>
<td>( h )</td>
</tr>
<tr>
<td>ordered iteration</td>
<td>( N \log N )</td>
<td>( N )</td>
<td>( N )</td>
</tr>
</tbody>
</table>

\( h \) = height of BST (proportional to \( \log N \) if keys inserted in random order)

Next lecture. Guarantee logarithmic performance for all operations.