### 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 URL with specified IP address.
- Given URL, find corresponding IP address.

---

#### 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>

<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>
Basic symbol table API

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

```java
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 paired with key
        (null if key is absent)
    }
    Value get(Key key) {
        value paired with key
        is there a value paired with key?
    }
    void delete(Key key) {
        remove key (and its value) from table
        is the table empty?
    }
    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
    }
}
```

**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);
}
```

**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.

**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.

**Built-in to Java (stay tuned)**

- `equals()` is `null`-safe.
- `hashCode()` is `null`-safe.

**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`, ...

**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;
    }
}
```

Check that all significant fields are the same.

Typically unsafe to use equals() with inheritance (would violate symmetry).

Optimize for true object equality.

Check for null.

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

Cast is guaranteed to succeed.

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()`.

```
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>6</td>
</tr>
<tr>
<td>A</td>
<td>7</td>
</tr>
<tr>
<td>M</td>
<td>8</td>
</tr>
<tr>
<td>P</td>
<td>9</td>
</tr>
<tr>
<td>L</td>
<td>10</td>
</tr>
<tr>
<td>E</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>12</td>
</tr>
</tbody>
</table>
**3.1 SYMBOL TABLES**

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

**ST test client for analysis**

**Frequency counter.** Read a sequence of strings from standard input and print out one that occurs with highest frequency.

```java
public class FrequencyCounter {
    public static void main(String[] args) {
        int minlen = Integer.parseInt(args[0]);
        String word = StdIn.readString();
        if (word.length() < minlen) continue;
        String max = "";
        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));
    }
}
```

**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.

**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 return lo;
    }
    return lo;
}
```

**Binary search in an ordered array**

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

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

```java
for (int i = 0; i < N; i++) {
    if (keys[i].compareTo(k) == 0) return i;
    else if (keys[i].compareTo(k) < 0) return i;
}
return N;
```

**Trace of binary search for rank in an ordered array**

```java
7  6  6    A  C  E  H  L  M  P   R  S  X
Trace of binary search for rank in an ordered array
```

**Binary search: trace of standard indexing client**

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

```java
for (int i = N; i > index; i--) {
    if (keys[i] == null) break;
    keys[i] = keys[i - 1];
} keys[index] = key;
```
3.1 Symbol Tables

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

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

Ordered symbol table API

```java
public class ST<Key extends Comparable<Key>, Value>
{
    create an ordered symbol table
    put(key-value pair into the table
    (remove key from table if value is null)
    value paired with key
    (null if key is absent)
    remove key (and its value) from table
    is there a value paired with key?
    is the table empty?
    number of key-value pairs
    smallest key
    largest key
    largest key less than or equal to key
    smallest key greater than or equal to key
    number of keys less than key
    key of rank k
    delete smallest key
    delete largest key
    number of keys in [lo..hi]
    keys in [lo..hi], in sorted order
    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>$\lg N$</td>
</tr>
<tr>
<td>insert / delete</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>$\lg N$</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>$N \lg N$</td>
<td>$N$</td>
</tr>
</tbody>
</table>

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

---

### 3.2 Binary Search Trees

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

#### 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.
**BST representation in Java**

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

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

**BST implementation (skeleton)**

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

    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;
        }
    }
}
```

**Binary search tree demo**

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

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

**BST insert: Java implementation**

```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;
}
```

**Cost.** Number of compares is equal to 1 + depth of node.

**Tree shape**

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

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

**Ex.** Insert keys in random order.

![BST insertion diagram]

**Correspondence between BSTs and quicksort partitioning**

![Correspondence diagram]

**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.

**Proposition.** [Reed, 2003] If \( N \) distinct keys are inserted in random order, expected height of tree is \( \sim 4.311 \ln N \).

![How Tall is a Tree?]

**But...** Worst-case height is \( N \).
(exponentially small chance when keys are inserted in random order)

ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Ordered Ops?</th>
<th>Operations on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
<td>equals()</td>
</tr>
<tr>
<td>(unordered list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>( \lg N )</td>
<td>( \lg N )</td>
<td>yes</td>
<td>compareTo()</td>
</tr>
<tr>
<td>(ordered array)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>1.39 ( \lg N )</td>
<td>next</td>
<td>compareTo()</td>
</tr>
</tbody>
</table>
3.2 Binary Search Trees

- BSTs
- ordered operations
- deletion

Minimum and maximum

Minimum. Smallest key in table.
Maximum. Largest key in table.

Q. How to find the min / max?

Floor and ceiling

Floor. Largest key ≤ a given key.
Ceiling. Smallest key ≥ a given key.

Q. How to find the floor / ceiling?

Computing the floor

Case 1. [k equals the key at root]
The floor of k is k.

Case 2. [k is less than the key at root]
The floor of k is in the left subtree.

Case 3. [k is greater than the key at root]
The floor of k is in the right subtree
(if there is any key ≤ k in right subtree); otherwise it is the key in the root.
Computing the floor

```java
public Key floor(Key key) {
    Node x = floor(root, key);
    if (x == null) return null;
    return x.key;
}
private Node 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);
    Node t = floor(x.right, key);
    if (t != null) return t;
    else return x;
}
```

Subtree counts

In each node, we store the number of nodes in the subtree rooted at that node; to implement size(), return the count at the root.

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

BST implementation: subtree counts

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

Remark. This facilitates efficient implementation of rank() and select().

Rank

Rank. How many keys < 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 return size(x.left);
}
```
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.

### 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>lg 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>lg N</td>
<td>h</td>
</tr>
<tr>
<td>rank</td>
<td>N</td>
<td>lg 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>

order of growth of running time of ordered symbol table operations

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

### 3.2 BINARY SEARCH TREES

- BSTs
- ordered operations
- deletion

**ST implementations: summary**

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>insert</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>sequential search (linked list)</td>
<td>N</td>
<td>N</td>
<td>N/2</td>
<td>no</td>
</tr>
<tr>
<td>binary search (ordered array)</td>
<td>lg N</td>
<td>lg N</td>
<td>N/2</td>
<td>yes</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
<td>yes</td>
</tr>
</tbody>
</table>
BST deletion: lazy approach

To remove a node with a given key:
- Set its value to null.
- Leave key in tree to guide search (but don’t consider it equal in search).

Cost. \( \sim 2 \ln N' \) per insert, search, and delete (if keys in random order), where \( N' \) is the number of key-value pairs ever inserted in the BST.

Unsatisfactory solution. Tombstone (memory) overload.

Deleting the minimum

To delete the minimum key:
- Go left until finding a node with a null left link.
- Replace that node by its right link.
- Update subtree counts.

public void deleteMin()
{  root = deleteMin(root);  }

private Node deleteMin(Node x)
{
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.count = 1 + size(x.left) + size(x.right);
    return x;
}

Hibbard deletion

To delete a node with key \( k \): search for node \( t \) containing key \( k \).

Case 0. [0 children] Delete \( t \) by setting parent link to null.

Case 1. [1 child] Delete \( t \) by replacing parent link.
Hibbard deletion

To delete a node with key $k$: search for node $t$ containing key $k$.

Case 2. [2 children]
- Find successor $x$ of $t$.
- Delete the minimum in $t$’s subtree.
- Put $x$ in $t$’s spot.

Deletion in a BST

First find the minimum.

Go right, then go left until reaching null.

Left link

Hibbard deletion: Java implementation

```java
public void delete(Key key) {
    root = delete(root, key);
}

private Node delete(Node x, Key key) {
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = delete(x.left, key);
    else if (cmp > 0) x.right = delete(x.right, key);
    else {
        if (x.right == null) return x.left;
        if (x.left == null) return x.right;
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    return x;
}
```

Hibbard deletion: analysis

Unsatisfactory solution. Not symmetric.

Surprising consequence. Trees not random (!) ⇒ $\sqrt{N}$ per op.

Longstanding open problem. Simple and efficient delete for BSTs.

ST implementations: summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Guarantee</th>
<th>Average Case</th>
<th>Ordered Iteration?</th>
<th>Operations on Keys</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>search</td>
<td>insert</td>
<td>delete</td>
<td>search</td>
</tr>
<tr>
<td>sequential search</td>
<td>$N$</td>
<td>$N$</td>
<td>$N$</td>
<td>$N/2$</td>
</tr>
<tr>
<td>(linked list)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>binary search</td>
<td>$\log N$</td>
<td>$N$</td>
<td>$N$</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>$N$</td>
<td>$1.39 \log N$</td>
</tr>
</tbody>
</table>

Next lecture. Guarantee logarithmic performance for all operations.