3.1 Symbol Tables

- API
  - sequential search
  - binary search
  - 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.

<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>

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>

Symbol table API

Associative array abstraction. Associate one value with each key.

```
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
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
```

API for a generic basic symbol 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 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, ...

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, File, URL, Date, ...

User-defined implementations. Some care needed.

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, File, URL, Date, ...

User-defined implementations. Some care needed.

Implementing equals for user-defined types

Seems easy.

```java
public class Record {
    private final String name;
    private final long val;
    private final int id;
    ...
    public boolean equals(Record y) {
        Record that = y;
        return (this.val == that.val) &&
               (this.id  == that.id)  &&
               (this.name.equals(that.name));
    }
}
```
Implementing equals for user-defined types

Seems easy, but requires some care.

```java
public final class Record
{
    private final String name;
    private final long val;
    private final int id;
    ...

    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.id == that.id) && (this.name.equals(that.name));
    }
}
```

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

Best practices.

- Compare fields mostly likely to differ first.
- No need to use calculated fields that depend on other fields.

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 a primitive array, apply to each element

Best practices.

- Compare fields mostly likely to differ first.
- No need to use calculated fields that depend on other fields.

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));
}
```

Frequency counter. Read a sequence of strings from standard input and print one that occurs with highest frequency.
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));
    }
}
Binary search

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

Rank helper function. How many keys < k?

### Binary search: Java implementation

```java
class OrderedArray {
    private int N;
    private Key[] keys; // array of keys
    private Value[] vals; // array of values

    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 mid;
        }
        return lo;
    }
}
```

### Binary search: mathematical analysis

**Proposition.** Binary search uses \( \log_2 N \) compares to search any array of size \( N \).

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

\[
T(N) \leq T\left(\left\lfloor N/2 \right\rfloor \right) + 1
\]

left or right half

Recall lecture 2.
Binary search: trace of standard indexing client

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

<table>
<thead>
<tr>
<th>key value</th>
<th>S</th>
<th>E</th>
<th>A</th>
<th>R</th>
<th>C</th>
<th>M</th>
<th>P</th>
<th>L</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
</tbody>
</table>

| entries in red | were inserted |
| entries in gray | did not move |
| entries in black | moved to the right |
| circled entries are changed values |

Elementary ST implementations: frequency counter

<table>
<thead>
<tr>
<th>operations</th>
<th>costs</th>
</tr>
</thead>
<tbody>
<tr>
<td>5737</td>
<td>0</td>
</tr>
<tr>
<td>5737</td>
<td>0</td>
</tr>
<tr>
<td>14350</td>
<td>0</td>
</tr>
</tbody>
</table>

Challenge. Efficient implementations of both search and insert.
Ordered symbol table API

Examples of ordered symbol-table operations

keys values
min() 09:00:00 Chicago
09:00:03 Phoenix
09:00:11 Houston
get(09:00:13) 09:00:59 Chicago
09:01:10 Houston
09:03:13 Seattle
select(?) 09:10:25 Seattle
09:14:25 Phoenix
09:19:32 Chicago
09:19:46 Chicago
keys(09:15:00, 09:25:00) 09:22:10 Chicago
09:22:43 Seattle
09:22:54 Seattle
09:25:52 Chicago
09:26:23 Chicago
ceiling(09:30:00) 09:35:21 Chicago
09:36:14 Seattle
max() 09:37:44 Phoenix
size(09:15:00, 09:25:00) is 5
rank(09:10:25) is 7

Binary search: ordered symbol table operations summary

<table>
<thead>
<tr>
<th>operation</th>
<th>sequential</th>
<th>binary search</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>insert</td>
<td>I</td>
<td>N</td>
</tr>
<tr>
<td>min / max</td>
<td>N</td>
<td>I</td>
</tr>
<tr>
<td>floor / ceiling</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>rank</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>select</td>
<td>N</td>
<td>I</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

public class ST<Key extends Comparable<Key>, Value>
head (ST())
head (create an ordered symbol table)
void put(Key key, Value val)
head (put key-value pair into the table (remove key from table if value is null))
Value get(Key key)
head (value paired with key (null if key is absent))
void delete(Key key)
head (remove key (and its value) from table)
boolean contains(Key key)
head (is there a value paired with key?)
boolean isEmpty()
head (is the table empty?)
int size()
head (number of key-value pairs)
Key min()
head (smallest key)
Key max()
head (largest key)
Key floor(Key key)
head (largest key less than or equal to key)
Key ceiling(Key key)
head (smallest key greater than or equal to key)
int rank(Key key)
head (number of keys less than key)
Key select(int k)
head (key of rank k)
void deleteMin()
head (delete smallest key)
void deleteMax()
head (delete largest key)
int size(Key lo, Key hi)
head (number of keys in [lo..hi])
Iterable<Key> keys(Key lo, Key hi)
head (keys in [lo..hi], in sorted order)
Iterable<Key> keys()
head (all keys in the table, in sorted order)
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).

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.

```java
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)

```java
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 void delete(Key key) {
        /* see next slides */
    }
    public Iterable<Key> iterator() {
        /* see next slides */
    }
}
```

root of BST

root
a left link
two child
of root

Anatomy of a binary tree

Anatomy of a binary search tree

Binary search trees

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 search**

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

**BST search: Java implementation**

```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 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 depth of node.
BST trace: standard indexing client

Tree shape

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

Remark. Tree shape depends on order of insertion.

Observation. If keys inserted in random order, tree stays relatively flat.

Ex. Insert keys in random order.
Correspondence between BSTs and quicksort partitioning

Remark. Correspondence is 1-1 if array has no duplicate keys.

BSTs: mathematical analysis

Proposition. If keys are inserted 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 keys are inserted in random order, expected height of tree is $\sim 4.311 \ln N$.

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

ST implementations: frequency counter

ST implementations: summary
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 ≤ to a given key.

**Ceiling.** Smallest key ≥ to 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;
}
```

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

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.

```
node count N
```

BST implementation: subtree counts

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

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

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

```
node count N
```

Rank

Rank. How many keys < k?

Easy recursive algorithm (4 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;
}
```

```java
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>Operation</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>1</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>

- $h = \text{height of BST}$ (proportional to $\log N$ if keys inserted in random order)
- worst-case running time of ordered symbol table operations

---

### Inorder traversal

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

**Example: Inorder traversal**

```
inorder(S)
inorder(E)
inorder(A)
enqueue A
inorder(C)
enqueue C
inorder(E)
enqueue E
inorder(R)
inorder(H)
enqueue H
inorder(M)
enqueue M
print R
enqueue S
inorder(X)
enqueue X
```

- Recursive calls
- Queue
- Function call stack

---

### BSTs

- ordered operations
- deletion
ST implementations: summary

<table>
<thead>
<tr>
<th>implementation</th>
<th>guarantee</th>
<th>average case</th>
<th>ordered</th>
<th>iteration?</th>
<th>operations on keys</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential search (linked list)</td>
<td>N</td>
<td>N</td>
<td>N/2</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>N/2</td>
<td>yes</td>
</tr>
<tr>
<td>BST</td>
<td>N</td>
<td>N</td>
<td>1.39 lg N</td>
<td>1.39 lg N</td>
<td>yes</td>
</tr>
</tbody>
</table>

Next. Deletion in BSTs.

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.

```java
public void deleteMin() {
    root = deleteMin(root);
}
private Node deleteMin(Node x) {
    if (x.left == null) return x.right;
    x.left = deleteMin(x.left);
    x.N = 1 + size(x.left) + size(x.right);
    return x;
}
```

Hibbard deletion

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

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

BST deletion: lazy approach

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

Cost. \( 2 \ln N' \) 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 overload.

Hibbard deletion

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

**Case 0.** [0 children] Delete \( i \) by setting parent link to null.
Hibbard deletion

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

Case 1. [1 child] Delete $t$ by replacing parent link.

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

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;
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left;
    }
    x.N = size(x.left) + size(x.right) + 1;
    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.
Next lecture. **Guarantee** logarithmic performance for all operations.