Trees

a unique path from root to every element

- **root** (no parent)
- **height** length of longest path from root to leaf
- **nodes**
- **edge**
- **leaves** (no children)
Which of these are (not) trees? (And why?)

- **a**: Not a tree (contains a cycle)
- **b**: Not a tree (contains a cycle)
- **c**: Not a tree (contains a cycle)
- **d**: Tree
- **e**: Tree
- **f**: Tree
Ancestors
Descendants

subtree
Tree height
the length of the longest path from the root to a leaf

h = 2
h = 1
h = 1
h = 0
h = -1 (!)

The height of an empty tree is -1.
Binary trees

**structure constraint**: every node has at most two children
Preorder traversal

Visit the root, then preorder traverse the left subtree, then preorder traverse the right subtree
Inorder traversal

Inorder traverse the left subtree, then visit the root, then inorder traverse the right subtree
Postorder traversal

Postorder traverse the left subtree, then postorder traverse the right subtree, then visit the root.
Binary search trees (BSTs)

**Order constraint:** every parent is greater than all the nodes in its left subtree and less than all the nodes in the right subtree.
Balanced binary search trees

structure constraint: every subtree is about the same size as its sibling
Perfect trees

structure: all leaves are at the same level and every level is full

• Most trees aren’t perfect (why not?)
• But perfect trees are useful for analyzing balanced trees.
BST algorithm: find
BST algorithm: find

Given a BST values and a number i:

find(i, values):

If the tree is empty, return false.
Let key be the value at the root of the tree.
If key is i, return true.
If i < key, call find on the left subtree.
If i > key, call find on the right subtree.
BST algorithm: insert

Given a BST values and a number $i$:

$\text{insert}(i, \text{values})$:

Look for $i$ in values.

**Insert $i$ as a leaf** where it should be.
Designing and implementing a new data structure

Interface and implementation

- **Interface**
  
  Answers: what can this data structure do

- **Implementation: encoding**
  
  Answers: how the structure is stored, using existing data structures

- **Implementation: operations**

  Answers: how the structure provides its interface via algorithms over the encoding

It should be possible to replace the implementation without modifying the interface.

We’ll talk only about the interface for trees (but you have access to the code for the implementation).
## Our Racket trees: Interface

Inductive data structure, manipulated via constructors, accessors, and operations

<table>
<thead>
<tr>
<th>constructors</th>
<th>accessors</th>
<th>operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>put together</em></td>
<td><em>take apart</em></td>
<td><em>often recursive</em></td>
</tr>
<tr>
<td>empty-tree</td>
<td>(empty-tree? &lt;tree&gt;)</td>
<td>(size &lt;tree&gt;)</td>
</tr>
<tr>
<td>(make-leaf &lt;key&gt;)</td>
<td>(leaf? &lt;tree&gt;)</td>
<td>(height &lt;tree&gt;)</td>
</tr>
<tr>
<td>(make-tree &lt;key&gt; &lt;left&gt; &lt;right&gt;)</td>
<td>(root &lt;tree&gt;)</td>
<td>(find &lt;value&gt; &lt;tree&gt;)</td>
</tr>
<tr>
<td></td>
<td>(left &lt;tree&gt;)</td>
<td>(insert &lt;value&gt; &lt;tree&gt;)</td>
</tr>
<tr>
<td></td>
<td>(right &lt;tree&gt;)</td>
<td>(traverse-inorder &lt;tree&gt;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(traverse-preorder &lt;tree&gt;)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(traverse-postorder &lt;tree&gt;)</td>
</tr>
</tbody>
</table>

Our BSTs won’t be balanced.
What are some good test cases for trees?
; the number of nodes in the tree

(define (size tree)
  (Your response))
; the number of nodes in the tree
(define (size tree)
 (if (empty-tree? tree)
     0
     (+ 1
        (size (left tree))
        (size (right tree)))))

size
Worst-case analysis
How bad can it get?

Given a collection of size $N$ and an operation:
What’s the worst input for the operation?
How expensive is the operation, for that input? (cost = # of elements visited)

<table>
<thead>
<tr>
<th></th>
<th>find</th>
<th>insert</th>
<th>min</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tree</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary search tree (BST)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For trees (including unbalanced BSTs), the worst-case version of an $N$-element tree is a “stick” (i.e., a linked list).
### Worst-case analysis

How bad can it get?

Given a collection of size N and an operation:

What’s the worst input for the operation?

How expensive is the operation, for that input?  *(cost = # of elements visited)*

<table>
<thead>
<tr>
<th></th>
<th>find</th>
<th>insert</th>
<th>min</th>
<th>list elements in order</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>List</strong></td>
<td>$O(n)$ elements visited</td>
<td>$O(1)$ elements visited</td>
<td>$O(n)$ elements visited</td>
<td>$O(n \log n)$ elements visited</td>
</tr>
<tr>
<td><strong>Tree</strong></td>
<td>$O(n)$ elements visited</td>
<td>$O(1)$ elements visited</td>
<td>$O(n)$ elements visited</td>
<td>$O(n \log n)$ elements visited</td>
</tr>
<tr>
<td><strong>Binary search tree (BST)</strong></td>
<td>$O(n)$ elements visited</td>
<td>$O(n)$ elements visited</td>
<td>$O(n)$ elements visited</td>
<td>$O(n \log n)$ elements visited</td>
</tr>
<tr>
<td><strong>balanced</strong></td>
<td>$O(\log n)$ elements visited</td>
<td>$O(\log n)$ elements visited</td>
<td>$O(\log n)$ elements visited</td>
<td>$O(n)$ elements visited</td>
</tr>
</tbody>
</table>

For trees (including unbalanced BSTs), the worst-case version of an N-element tree is a “stick” (i.e., a linked list).
Analyze size, using a recurrence relation

For a given cost metric: additions; on the worst-case input: a stick

1. Translate the base case(s), using specific input sizes
   How many steps does this base case take?

2. Translate the recursive case(s), using input size N
   Define T(N) recursively, in terms of smaller cost.

   (define (size tree)
     (if (empty-tree? tree)
         0
         (+ 1 (size (left tree)) (size (right tree))))
   
   T(0) = 0
   T(N) = 1 + T(N-1) + 0

   T(N) = 1 + T(N-1) + 0
   = 1 + 1 + T(N-2)
   = 1 + 1 + 1 + T(N-3)
   ...
   = 1 + 1 + 1 + ... 1 + T(N-N)
   = 1*1 + T(N-1)
   = 2*1 + T(N-2)
   = 3*1 + T(N-3)
   ...
   = N*1 + T(N-N) = N ∈ O(N)