Binary Tree Properties

- Binary trees
- Binary tree properties
- Tree traversals
- Binary tree operations
Binary Trees

- A *binary tree* is a tree with an arity of 0, 1 or 2: a node in the tree may have 0, 1, or 2 children only.
- Some examples of binary trees:
Binary trees: some special cases

(a) A **full** binary tree of height 3.

(b) **Complete** binary trees of height 3 and 4.

**A full binary tree** of height $h$ is one in which all nodes from level 1 through level $h - 1$ have two children.

**A complete binary tree** of height $h$ is one in which all nodes from level 1 through $h - 2$ have two children and all the children of nodes at level $h - 1$ are contiguous and to the left of the tree.
Binary Tree properties

- Consider a “full” binary tree (every level that has any nodes at all has as many nodes as possible):
  - How many nodes at level 1? ____
  - How many nodes at level 2? ____
  - How many nodes at level 3? ____
  - Generalizing, how many nodes at level L? ____
  - And so, how many nodes in a full binary tree of height h? __________
  - And so, what is the height of a full binary tree with n nodes? __________
Binary Tree properties

• Generalizing, how many nodes at level $L$? $2^{L-1}$
• And so, how many nodes in a full binary tree of height $h$?

$$n = \sum_{L=1}^{h} 2^{L-1} = 2^h - 1$$

• And so... what is the height of a full binary tree of size $n$?

$$2^h - 1 = n$$
$$h = \log_2(n + 1)$$
$$h = \Theta(\log n)$$
Binary Tree properties

Property 10.1  A full binary tree of height $h$ has $2^h - 1$ nodes

Property 10.2  The height of a full binary tree with $n$ nodes is $\log_2 (n+1)$. (This is also the length of the longest path in a full binary tree.)

Property 10.3  The height of a binary tree with $n$ nodes is at least $\log_2 (n+1)$ and at most $n$ (this “worst case” occurs when no node has more than 1 child).
Tree traversals

- A traversal of a data structure visits each element of the structure, in some sequence

- For a linear data structure, a traversal typically visits elements from first to last

- But what about a hierarchical structure like a tree?

- Different sequences are possible

- The most common tree traversals are: **preorder**, **inorder**, **postorder**, and **level order** traversals
Tree traversals of binary trees

(Note: a tree traversal starts at the root of the tree.)

- **preorder** traversal: visit the current node; perform a preorder traversal of its left subtree; then perform a preorder traversal of the its right subtree

- **inorder** traversal: perform an inorder traversal of the left subtree of the current node; then visit the current node; then perform an inorder traversal of its right subtree

- **postorder** traversal: perform a postorder traversal of the left subtree of the current node; then perform a postorder traversal of its right subtree; then visit the current node

- **level order** traversal: visit the node at level 1; then visit the nodes at level 2, left to right; then visit the nodes at level 3, left to right; etc.
# Tree Traversals

<table>
<thead>
<tr>
<th>Traversal Order</th>
<th>Order in Which the Nodes Are Visited</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>preorder( node )</strong></td>
<td></td>
</tr>
<tr>
<td>if node is not null</td>
<td></td>
</tr>
<tr>
<td>visit this node</td>
<td></td>
</tr>
<tr>
<td>preorder( node’s left child )</td>
<td></td>
</tr>
<tr>
<td>preorder( node’s right child )</td>
<td></td>
</tr>
<tr>
<td><strong>preorder(root)</strong></td>
<td></td>
</tr>
<tr>
<td>visits the nodes in the following order: A B D E C F G</td>
<td></td>
</tr>
</tbody>
</table>

| **inorder( node )**   |
| if node is not null  |
| visit this node      |
| inorder( node’s left child ) |
| **inorder(root)**   |
| visits the nodes in the following order: D B E A F C G |

| **postorder( node )** |
| if node is not null  |
| visit this node     |
| postorder( node’s left child ) |
| postorder( node’s right child ) |
| **postorder(root)** |
| visits the nodes in the following order: D E B F G C A |

| **levelorder( node )** |
| if node is not null |
| add node to a queue  |
| while the queue is not empty |
| get the node at the head of the queue |
| visit this node      |
| if the node has children |
| put them in the queue in left to right order |
| **levelorder(root)** |
| visits the nodes in the following order: A B C D E F G |

Note the recursive nature of **preorder()**, **inorder()** and **postorder()**.
Binary Tree Operations

Common operations to implement on a binary tree structure:

• **add** a node at some position in the tree
• **remove** a node from a tree
• **traverse** a tree
• **navigate** around the tree: given a node, get to its parent and children
• **return** a **data** element contained in a node
To enable those operations, we can provide a binary tree node with four attributes:

- the data element to be stored in the node
- left child
- right child
- parent
class BinaryTreeNode<E> {
    E element;
    BinaryTreeNode<E> parent;
    BinaryTreeNode<E> leftChild;
    BinaryTreeNode<E> rightChild;
}

Having both parent and child pointers makes it easier to move up and down the tree – this is a hierarchical doubly linked list (sort of)
Removing a node from a binary tree

- Whether using an array or linked dynamic objects, all the required binary tree operations must be implemented.

- Here we will look briefly at the remove operation.

- The remove operation must preserve the binary tree invariants: at least, removing a node from a binary tree must result in a binary tree!

- Special cases of binary trees (heaps, binary search trees, etc.) will have additional invariants that must be preserved.
Binary Tree: Remove()

- Removing a leaf node is simple (not shown)
- Removing an interior node is a bit trickier (see below)

(a) Target node A is an internal node; find a leaf in one of its subtrees

(b) Replace target’s element with the *element* from a leaf descendant; remove the leaf
public void remove( BinaryTreeNode<E> target ) {

if target is null there is nothing to do, so return
if ( target == null )
    return; // nothing to remove
BinaryTreeNode<E> node = target;

if target is internal
    // If this is an internal node, we need to get another
    // node (a leaf descendant) to put in its place
    if ( node.isInternal () ) {
        get a leaf descendant of target; move leaf ‘s element into target
        and detach the leaf
        BinaryTreeNode<E> leaf = getaLeafDescendant( node )
        node.setElement( leaf.element() );
    }
else target is a leaf and needs to be detached
else
    detachFromParent( node );
decrement tree’s size by 1
    size--;
}
Testing the Implementation

• As always: understand the required behavior of any software, write a test plan, then code up and run the tests

• A challenge for testing a tree implementation is ensuring that elements are in the correct position within the tree following and add or delete

• Helpful idea: Use tree traversal to create a list of the tree’s elements, and compare the list against what is expected
Next time

- A Binary Search Tree ADT
- A Linked Implementation of Binary Search Tree
- Binary Search Tree Operation Time Costs
- The Importance of Being Balanced
- AVL trees

Reading: Gray, Ch 10