AUBURN

UNIVERSITY

SAMUEL GINN
COLLEGE OF ENGINEERING

Binary Trees

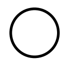
Binary Trees

Binary trees are trees of order 2.

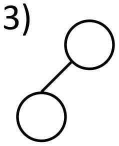
Examples…

1)

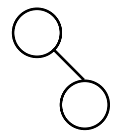
2)



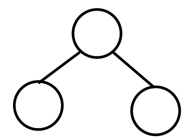
3)



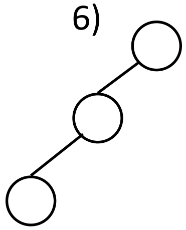
4)



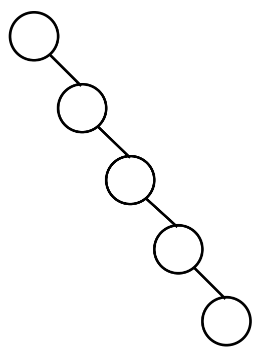
5)



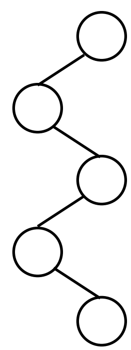
6)



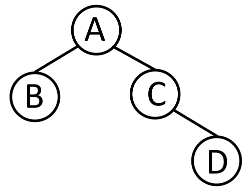
7)



8)



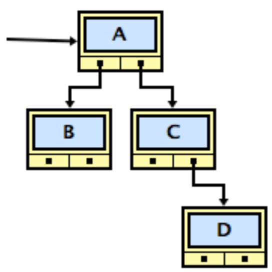
Implementation strategies



Node-and-link based

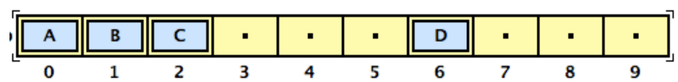
```
class BTN<T>
{
   T element;
   BTN left;
   BTN right;
}
```

This implementation matches our conceptual picture of what a tree looks like.



Array based

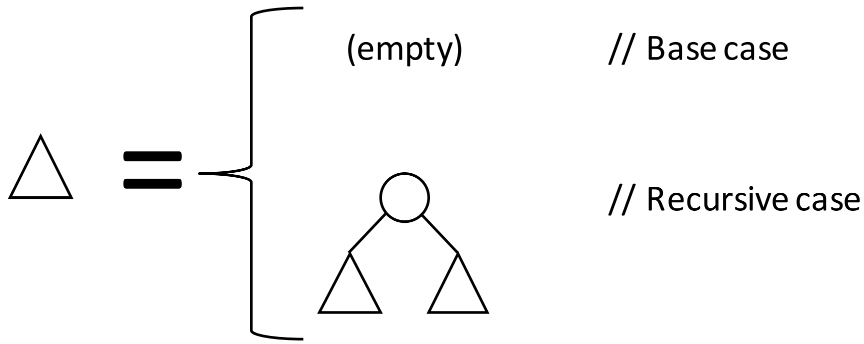
```
-Store the root at index 0
-For a node stored at index i
    -Left child at 2i + 1
    -Right child at 2i + 2
    -Parent at (i-1)/2
```



This implementation could use far too much space. Think about a right skewed tree …

Recursive definition

A binary tree is a tree that is either empty or it is a single node that has two binary trees as its left and right subtrees.




```
if (isEmpty()) {
   // do something trivial
} else {
   // In some order:
   // do something with the node
   // recursively process the left subtree
   // recursively process the right subtree
}
```

Computing height

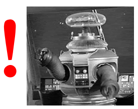
Height = length of the longest path from a given node to a descendent leaf

Think recursively…

Base case

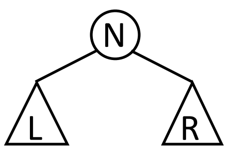
(empty) No height (height = 0)

!



Some define the height of an empty tree as -1. This makes no intuitive sense; our way is better.

Recursive case



The node (N) contributes 1 to the height

Calculate the height of the left subtree (L)

Calculate the height of the right subtree (R)

Height of this node is 1 + maximum of h(L) and h(R)

Computing height

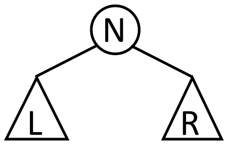
Height = length of the longest path from a given node to a descendent leaf

Think recursively…

Base case

(empty)

Recursive case




```
int height(Node n) {
    if (n == null) {
        return 0;
    } else {
        int leftHeight = height(n.left);
        int rightHeight = height(n.right);
        return 1 + Math.max(leftHeight, rightHeight);
    }
}
```

Computing height

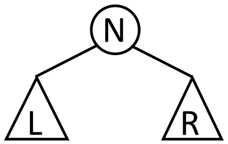
Height = length of the longest path from a given node to a descendent leaf

Think recursively…

Base case

(empty)

Recursive case




```
int height(Node n) {
    if (n == null) {
        return 0;
    }
    int leftHeight = height(n.left);
    int rightHeight = height(n.right);
    return 1 + Math.max(leftHeight, rightHeight);
}
```

Searching in a tree

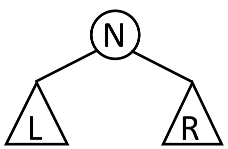
Search the tree for a particular element. Return true if the value is found, false otherwise.

Think recursively…

Base case

(empty)

Recursive case




```
boolean search(Node n, Object target) {
    if (n == null) {
        return false;
    }
    if (n.element.equals(target)) {
        return true;
    }
    boolean found = search(n.left, target);
    if (!found) {
        found = search(n.right, target);
    }
    return found;
}
```

Traversing a tree

Systematically visit each node in the tree.

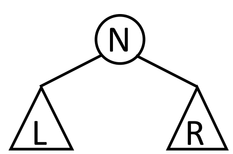
Think recursively…

Base case

(empty) Nothing to traverse

Since there's no action to take in the base case, let the if statement check for it **not** being the base case.

Recursive case



In some order:
- visit the root node of the subtree (N)
- recursively visit the left subtree (L)
- recursively visit the right subtree (R)

NLR
NRL
LNR
LRN
RNL
RLN

Traversing a tree

Systematically visit each node in the tree.

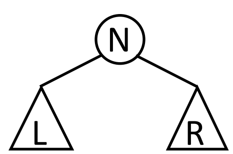
Think recursively…

Base case

(empty) Nothing to traverse

Since there's no action to take in the base case, let the if statement check for it **not** being the base case.

Recursive case



In some order:
- visit the root node of the subtree (N)
- recursively visit the left subtree (L)
- recursively visit the right subtree (R)

NLR Preorder
NRL
LNR Inorder
LRN Postorder
RNL
RLN

Traversing a tree

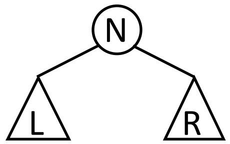
Systematically visit each node in the tree.

Think recursively…

Base case

(empty)

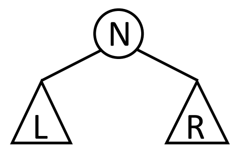
Recursive case




```
void preorder(Node n) {
   if (n != null) {
      visit(n);
      preorder(n.left);
      preorder(n.right);
   }
}
```

Binary tree traversals

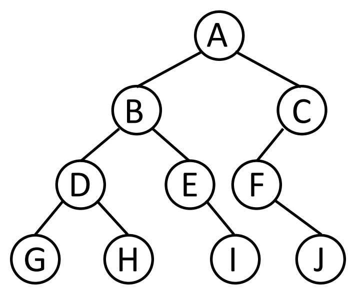
Recursive Case…



Preorder: NLR

Postorder: LRN

Inorder: LNR



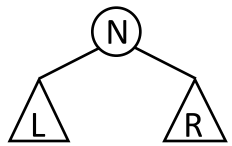
Preorder: A B D G H E I C F J

Postorder: G H D I E B J F C A

Inorder: G D H B E I A F J C

Binary tree traversals

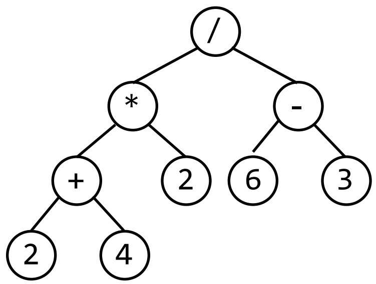
Recursive Case…



Preorder: NLR

Postorder: LRN

Inorder: LNR



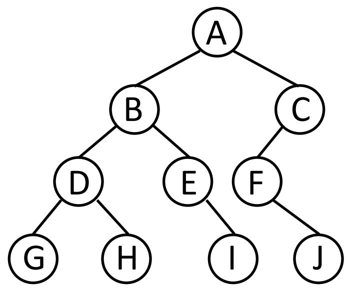
Preorder: / * + 2 4 2 – 6 3

Postorder: 2 4 + 2 * 6 3 - /

Inorder: 2 + 4 * 2 / 6 - 3

Level order

Preorder, inorder, and postorder are all **depth-first** strategies.
A **breadth-first** strategy would visit the nodes level by level
(i.e., top to bottom, left to right).



Level-order (breadth-first) traversal

```
Let q be an initially empty FIFO queue.
q.enqueue(root);
while (q is not empty) {
   n = q.dequeue();
   visit(n);
   if (n has a left child) {
      q.enqueue(n.left);
   }
   if (n has a right child) {
      q.enqueue(n.right);
   }
}
```

If "visit" prints the node elements, then the output for this tree would be:
A B C D E F G H I J