Binary Trees

COMP 2210 – Dr. Hendrix

AUBURN UNIVERSITY

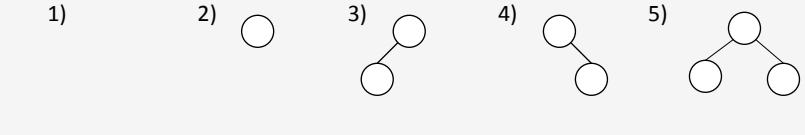
SAMUEL GINN COLLEGE OF ENGINEERING

Binary Trees

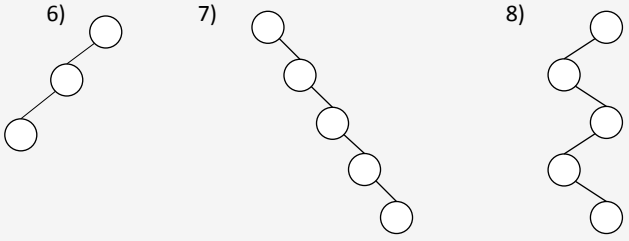
Binary trees are trees of order 2.

Examples …

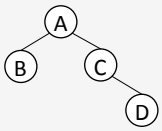
1) 2) 3) 4) 5)



6) 7) 8)



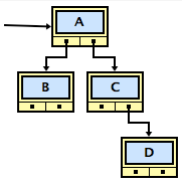
Implementation Strategies



Node-and-link based

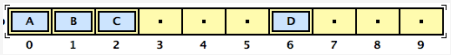
```
class BTN<T>
{
   T element;
   BTN left;
   BTN right;
}
```

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



Array based

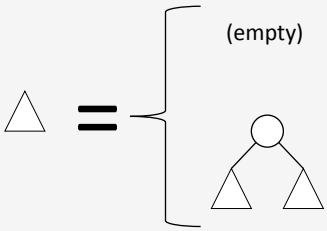
```
-Store the root at index 0
-For a node stored at index i
    -Left child at 2i + 1
    -Right child at 2i + 2
    -Parent at (i-1)/2
```



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.

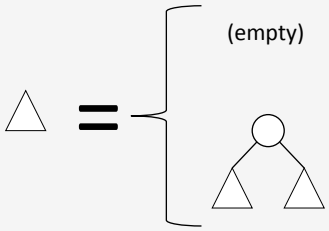



```
if (isEmpty())
{
   // do something trivial
}
else
{
   // In some order:
   // do something with the node
   // recursively process the left subtree
   // recursively process the right subtree
}
```

Common algorithms on binary trees

- Calculating the height of a node
- Calculating the number of nodes in a subtree
- Searching for a value in the tree
- Traversing the tree

Think recursively!



Computing height

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

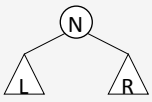
Think recursively…

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



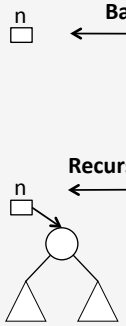
The node (N) contributes 1 to the height

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



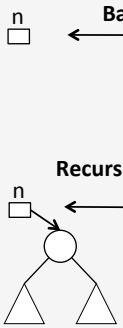

```
public int height(BTN n)
{
   if (n == null)
   {
      return 0;
   }

   else
   {
      int lh = height(n.getLeft());
      int rh = height(n.getRight());
      return 1 + max(lh, rh);
   }

}
```


Searching a binary tree

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




```
public boolean search (BTN n, T target)
{
   if (n == null)
   {
      return false;
   }

   else
   {
      if (n.getElement().equals(target))
         return true;
      else {
         boolean found = search(n.getLeft(), target);
         if (!found)
            found = search(n.getRight(), target);
         return found;
       }
   }

}
```

Binary Tree Traversal

Walk over the entire tree and "visit" each node once.

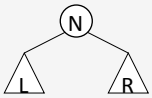
Think recursively…

Base case

(empty) — Nothing to traverse

Since there's nothing to do in the base case, we can just check for the recursive case.

Recursive case



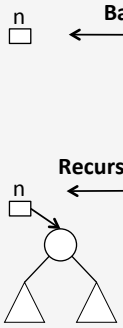
Visit the node (N)

Traverse the left subtree

Traverse the right subtree

*Only one of the possible orderings. This is called **preorder**, since the node is visited prior to either subtree.*

Binary Tree Traversal - Preorder



Base Case

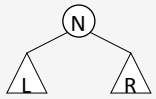
```
public void preorder(BTN n)
{
   if (n == null)
   {
      ;
   }
   else
   {
      visit(n);
      preorder(n.getLeft());
      preorder(n.getRight());
   }
}
```

Since there's nothing to do in the base case, we can just check for the recursive case.

```
if (n != null)
{
   visit(n);
   preorder(n.getLeft());
   preorder(n.getRight());
}
```

Binary Tree Traversals

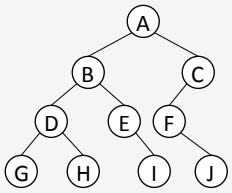
Recursive Case…



Preorder: NLR

Postorder: LRN

Inorder: LNR



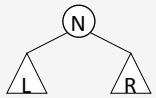
Preorder: A B D G H E I C F J

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

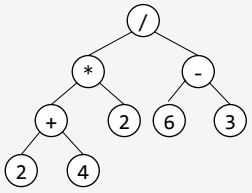
Recursive Case…



Preorder: NLR

Postorder: LRN

Inorder: LNR



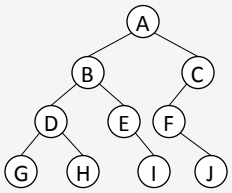
Preorder: / * + 2 4 2 – 6 3

Postorder: 2 4 + 2 * 6 3 - /

Inorder: 2 + 4 * 2 / 6 - 3

Binary Tree Traversal – 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