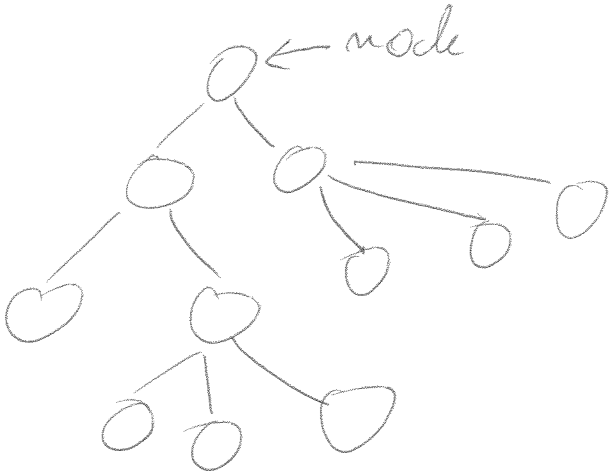
COMP-250 – Lecture 24: Trees: Binary trees

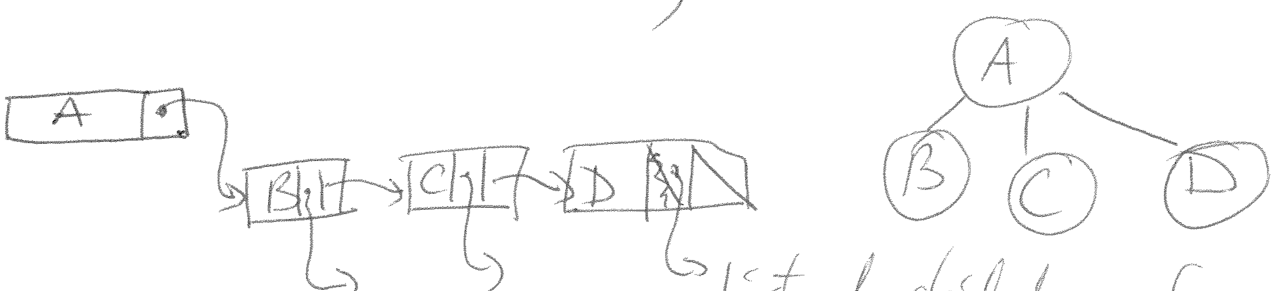


Each node has exactly 1 parent, except root (unique) has 0 parents.

Tree: made up of Nodes ⇒ Node class
↳ root

≈ List: Node, List has a ref to head

```
public class Node<T> {
    T content; //stuff node has)
    List<Node> children; //holds all the children
```



list of children for each of these nodes

```
public class Tree<T> {
    Node<T> root;

    // Methods
    Traversal
    // Insert / remove content
```

Pre-order: process crt node __before__ its descendents

Post-order: process node __after__ all its descendents.

Pre-order - typical for processing game playing trees

__Algorithm__ Pre-order (Node n)

```
process (n);
for each c ∈ n.children
    Preorder (c)
```

Post-order traversal:

```
for each c ∈ n.children
    Postorder (c)
process (n)
```

Preorder (root) or Postorder (root)

Intuitively: $O(n)$ where n is number of nodes in tree

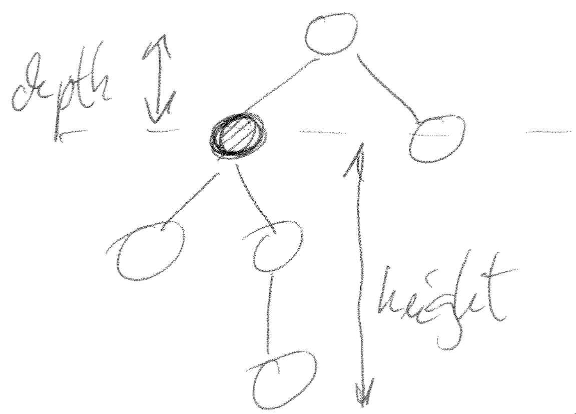
n nodes

$$T(n) = \underbrace{T_{process}(root)}_{(O(1))\ c} + \sum_{c \in root.children} T(n_c)$$

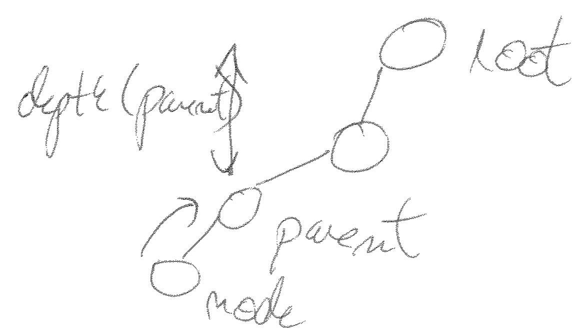
n_c = number of nodes under child c

Notice: $n = 1 + \sum_c n_c$

To make algs that are faster, we'd like to eliminate the sum $\Rightarrow$ look at particular sub-trees



depth / height of a node

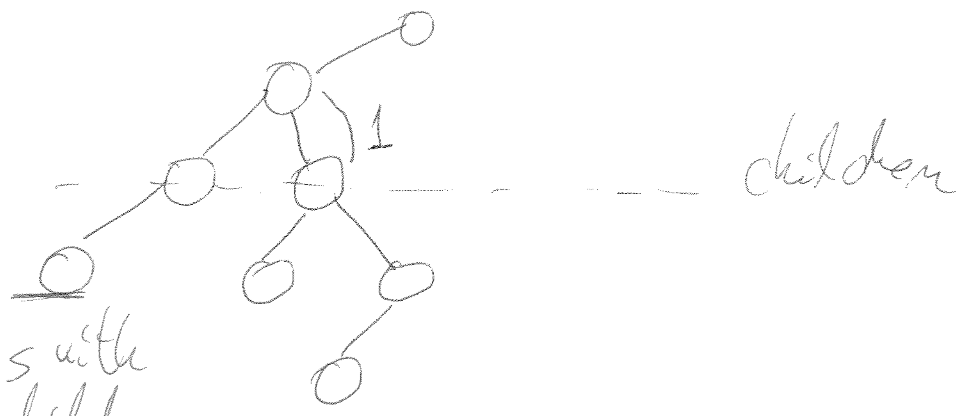


Depth of a node (recursively):

$depth(root) = 0$

$depth(node) = 1 + depth(parent)$

Height of a node



Nodes with no children (leaf)

$$\text{height}(\text{leaf}) = 0$$

$$\text{height}(\text{node}) = 1 + \max_{c \in \text{node.children}} \text{height}(c)$$

<u>Binary Trees</u>: each node has at most 2 children

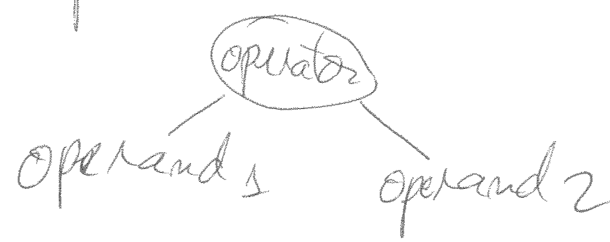
```
public class Node<T> {
    T content;
    Node<T> left, right;  // NULL indicates no child
    public class Node(   ) → initialize content
            left = null;
            right = null;
  get/set methods for all relevant fields
```

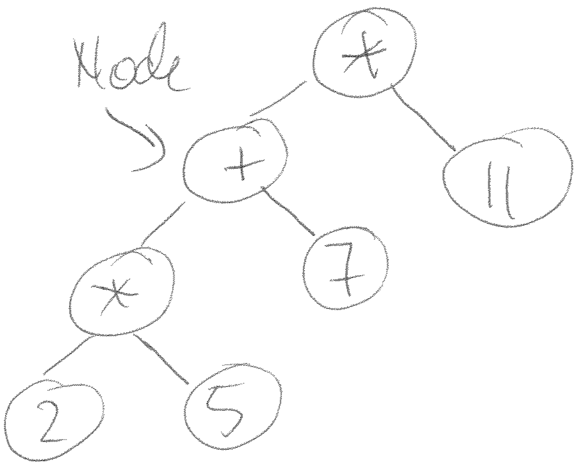
```
public class BinaryTree<T> {
    Node<T> root;
    Pre-order
    Post-order traversal

E.g. Preorder (~~Node~~ node) {
    process(node)        // base case for null
    Preorder(left)
    Preorder(right);
 }
```

In-order traversal:

(2 * 5 + 7) * 11 ⟹ Expression Tree





Computing value: traversal of tree.

compute children; then do operation (post-order)

construct tree: left subtree; parent; right subtree (in-order)

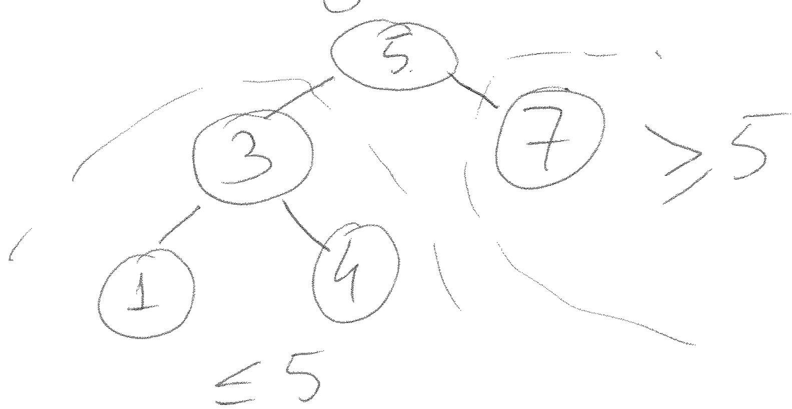
L24-6

In-order Traversal:

inorder (left)   O(n)
process (content)
inorder (right)   only looking in 1 subtree

Binary Search Trees (BST)

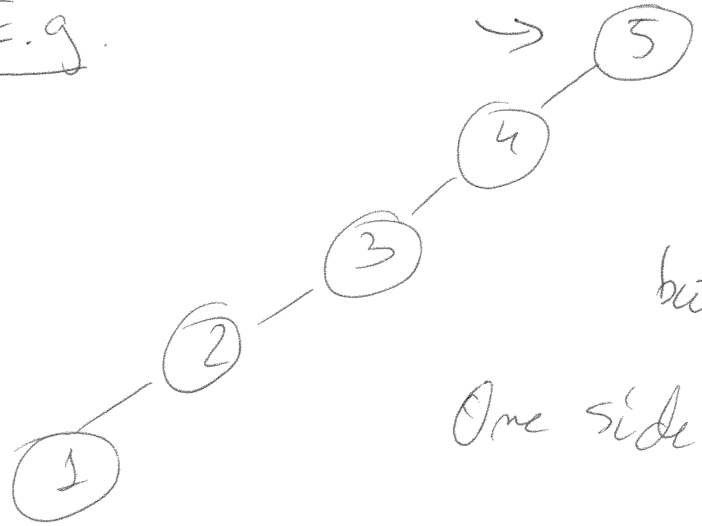
Goal: data structure that allows fast searches
Type of content at the node needs to be Comparable
(to allow ordering)



At any node, left subtree content is smaller
right subtree content is larger.

Binary search: content to find
- compare the ct node; if equal return true
- else either left or right but not both!

E.g.



"Degenerate" case:
it is a binary search tree
but imbalanced

One side has - more nodes
- larger hight of nodes

Balanced binary search trees (different conditions)

E.g. $|\text{height}(\text{left}) - \text{height}(\text{right})| \le 1$

$|\text{nrnodes}(\text{left}) - \text{nrnodes}(\text{right})| \le 1 \Rightarrow$ more stringent

Under balance: searching in $O(\log_2 n)$

Insert / remove: make sure the BST condition is met + balance condition is met