Binary search trees are **recursive data structures**

* Each node is a root of 0,1,2 sub-trees
* LHS values = **smaller** **than root**
* RHS values = **larger than root**

Operations on trees:

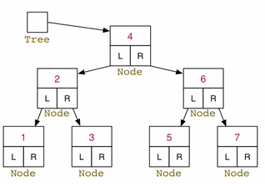
* Insert(Tree, item) / delete(Tree, item) / search(Tree, item)
* Housekeeping = New(); dispose(); show(); empty();

Nodes contain Items. We just show **Item.key.**

Shape of trees is determined by order of insertion (balanced vs non-balanced). We want trees to be as shallow as possible.

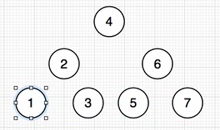
**LEVEL of node** = path length from ROOT 🡪 NODE

**DEPTH of tree** = max path length from ROOT 🡪 LEAF

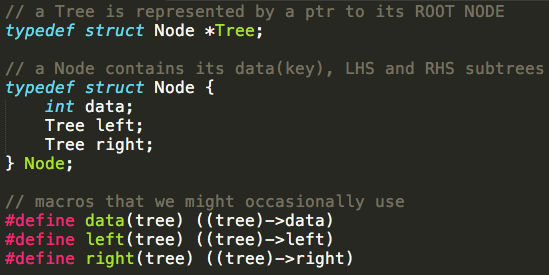
* Depth of tree with N nodes:
  + Min depth = **floor(log2n)**
  + Max depth = **n-1**
* Height balanced tree: **depth(LHS subtree)** = **depth(RHS subtree)**
* Time complexity of tree = **O(depth)**

Insertion into BSTs:

* Example: 4 2 6 5 1 7 3



Representation of BSTs:

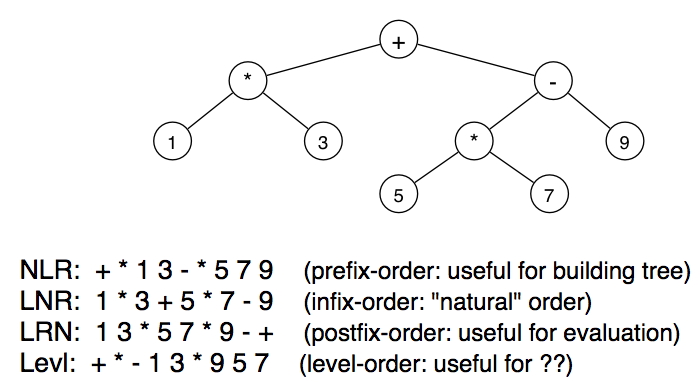


Binary Tree Traversal

* **PREfix-order (NLR)** = Root 🡪 Left Subtree 🡪 Right Subtree
* **INfix-order (LNR)** = Left Subtree 🡪 Root 🡪 Right Subtree
* **POSTfix-order (LRN)** = Left Subtree 🡪 Right Subtree 🡪 Root
* **Level-order** = Root 🡪 All children 🡪 Next level of children

Traversing an Expression Tree

* Character values as keys (rather than integers as keys)



**Prefix (NLR):** Great for reconstructing a tree. (Dump & Rebuild)

**Infix (LNR):** “natural” order

**Postfix (LRN):** Useful for evaluation

* Work you way down to leaf (number val) (E.g. #1)
* Push leaf onto stack
* Find corresponding leaf for operator + push on stack (E.g. #3)
* Apply operator **\*** to two items at top of the stack, giving  
  a new value to push onto the stack (parent of two leaves)

**Level-order**: Useful for ??

Binary Tree Traversal