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:

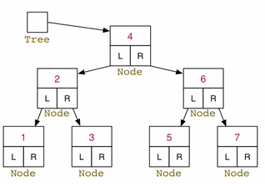
* Insert(Tree, item) / delete(Tree, item) / search(Tree, item)
* Housekeeping = New(); dispose(); show(); empty();

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.

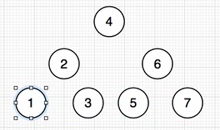
**LEVEL of node** = path length from ROOT 🡪 NODE

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

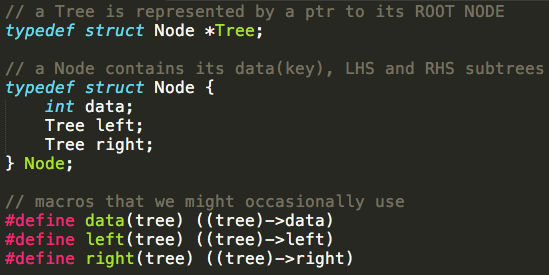
* Depth of tree with N nodes:
  + Min depth = **floor(log2n)**
  + Max depth = **n-1**
* Height balanced tree: **depth(LHS subtree)** = **depth(RHS subtree)**
* Time complexity of tree = **O(depth)**

Insertion into BSTs:

* Example: 4 2 6 5 1 7 3



Representation of BSTs:

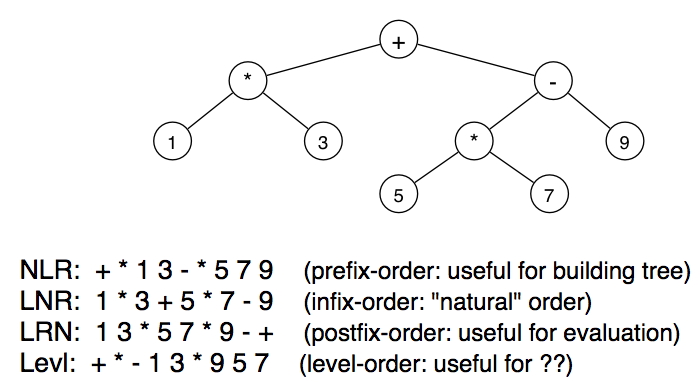


Binary Tree Traversal

* **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

* Refer to code

Binary Tree Removal is harder

* Four cases to consider:
  + **Empty Tree** = new tree is also empty
  + **Node w/ Zero Subtrees** = **unlink node from parent**
  + **Node w/ One Subtree** = replace by **child node**
  + **Node w/ Two Subtrees** = replace by **successor**

Successor Node = The smallest value in the RHS Subtree (that is larger than the Root Node)  
Make the Successor Node the new Root Node  
We choose the RHS Subtree as every node on the LHS Subtree will be smaller than the successor

* Refer to code

Binary Search Tree ADT

* Data type simply called **Tree**

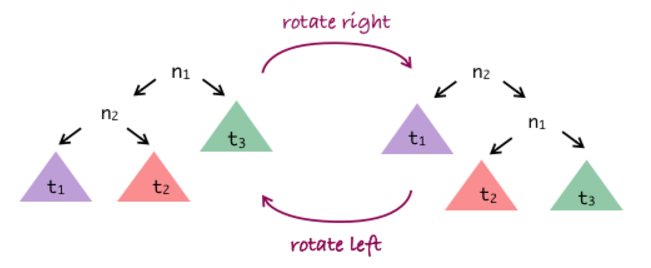
Generating values in **Prefix order**

* Inserting values in the “correct” order will ensure tree balance
* Write a function that generates prefix order:
  + Generates values in range **lo 🡪 hi**
  + First = mid-point, second = mid of lower-half
  + Store values in array **v[0 .. N – 1 ]**

**Balanced Binary Search Trees**

* Goal: Build binary search trees w/ **minimum depth** **= minimum worst case search cost**
* Perfectly balanced tree has:
  + **Abs(nnodes(LHS) – nnodes(RHS)) < 2 for every node**
    - Same number of nodes on each side
  + Depth of **log2n 🡪 worst case search O(Logn)**
* Three strategies to improve worst case search in BSTs:
  + **Randomise** = reduce chance of worst-case scenario  
    (similar to pre-randomising values for QuickSort, as pre-sorted values will be worst-case)
  + **Amortize** = do more work at insertion to make search faster
  + **Optimize** = implement all operations with performance bounds

**Operations for Rebalancing**



How does rebalancing help?

* E.g. if we discovered that LHS Subtree has a lot more nodes than RHS Subtree and is unbalanced, a rotation may help rebalance.
* Rotation helps with a lot of different situation
* One way it helps, is through insertion at root

**Insertion at Root**

* Method for insertion at root:
  + BASE CASE:
    - Tree is empty, make new node and make it the root
  + RECURSIVE CASE:
    - Insert new node as root of L/R subtree
    - Lift new node to root by R/L rotation
* Example of **Insertion of Node(Key = 4) at Root**

