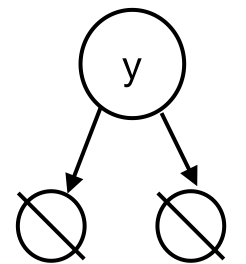
#### **Number of null pointers in a Binary Tree**

* + **Theorem:** A binary tree with **n** data items has **n+1** null pointers.
  + *Proof.* Let **NULL**(n) be the number of null pointers in a tree with **n** nodes.
    - Target: **NULL**(n) = n+1, for **n** >= 0
    - Proof by induction:
    - Base case:
      * when n=0, **NULL**(0) = 1, Just a empty tree with one null pointer: root = nullptr



* when n=1, **NULL**(1)
* = 2

when n=2, **NULL**(2) = 3

* + - **Induction step:**
      * Inductive hypothesis: **NULL**(j) = j+1, for any j<n
      * Target: for tree **T** with **n** nodes:

**NULL**(n) = n+1

* + - * Look at the root:

#nodes in left subtree + #nodes in right subtree = n-1

* + - * Suppose left subtree has **q** nodes, right subtree has **n-q-1** nodes
      * Then, **q<n**, and **n-q-1<n**
      * Then, since the root has no null pointers,

**NULL**(n) = **NULL**(q) + **NULL**(n-q-1) = q+1 + n-q-1+1 = n+1

* + **Observation**: the overhead of the tree data is proportional to the number of nodes. Since NULLs don’t take up a lot of memory, this is very good.

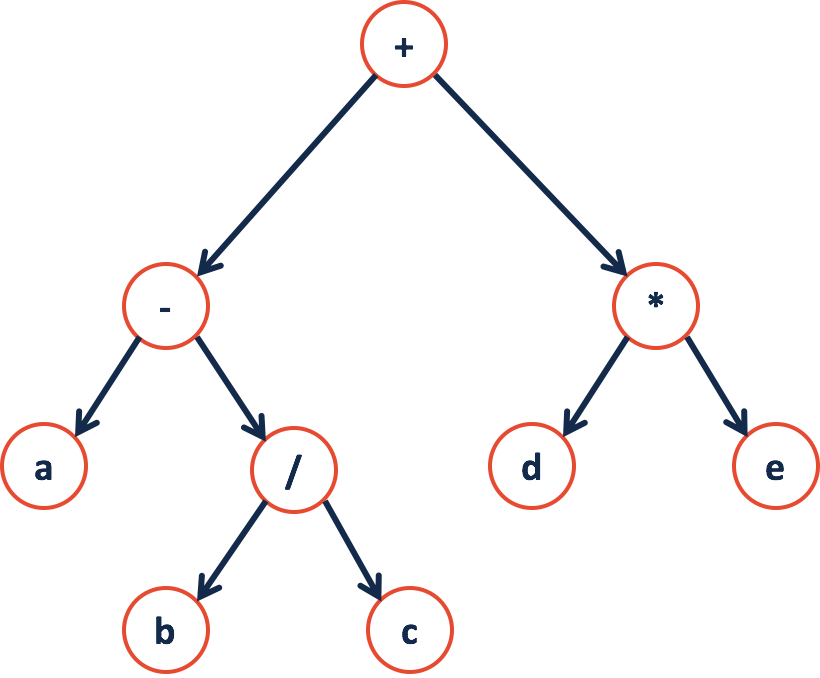
Simpler proof: (2n-(n-1)) = n+1, where 2n is the number of edges if the binary tree were full; (n-1) is the number of existing edges in the tree with n nodes.

#### **Traversal**

* + To traverse a tree means we process every element exactly once in a tree
    - **Pre-order:** process the data first, then left child, then the right child
    - **In-order:** left child, process the data, right child
    - **Post-order:** left child, right child, process the data the last

|  |  |
| --- | --- |
| BinaryTree.cpp | |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | void BinaryTree<T>::preOrder(TreeNode \* cur) {  if (cur != NULL) {  yell(cur->data); // yell is some imaginary function  preOrder(cur->left);  preOrder(cur->right);  }  }  void BinaryTree<T>::inOrder(TreeNode \* cur) {  if (cur != NULL) {  inOrder(cur->left);  yell(cur->data);  inOrder(cur->right);  }  }  void BinaryTree<T>::postOrder(TreeNode \* cur) {  if (cur != NULL) {  postOrder(cur->left);  postOrder(cur->right);  yell(cur->data);  }  } |

#### **In-order print out of the tree**



* + Recursion all the way to the left, print out **a**, then **-**
  + Then to **b / c;** then **+, d, \*, e;**
  + It will be **a - b / c + d \* e!**
  + Tree is used a lot in parsing - in this example, a language with **binary operators** is parsed into a **binary tree**
  + Therefore, if we use different methods of traversals we will have different outputs and meanings.

#### **Level Order Traversal**:

#### We use it to traverse the tree by every level

* + We will use an iterative approach, by using a queue to keep track of each node we encounter on each level

|  |
| --- |
| 1. Enqueue the root 2. While queue is not empty    1. Dequeue **e**    2. Yell **e**->data    3. Enqueue(**e**->left),   Enqueue(**e**->right) |