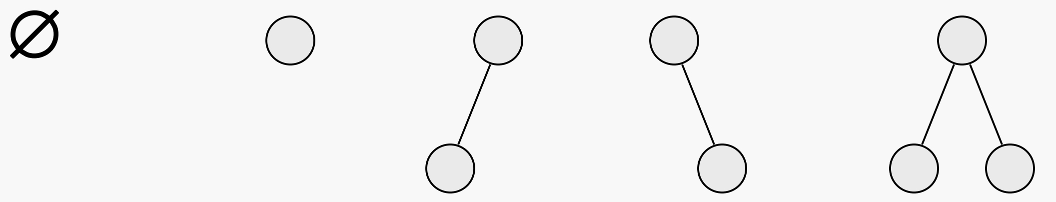
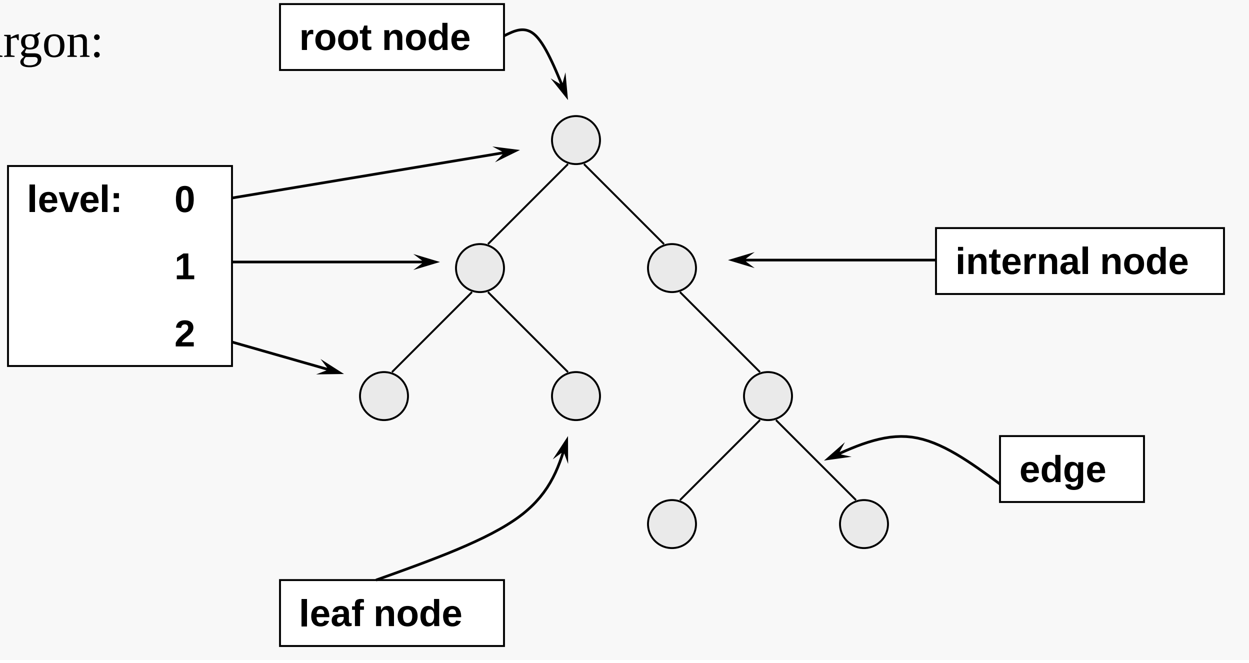
Binary Trees

A binary tree is either empty, or it consists of a node called the root together with two binary trees called the left subtree and the right subtree of the root, which are disjoint from each other and from the root.

For example:



Jargon:

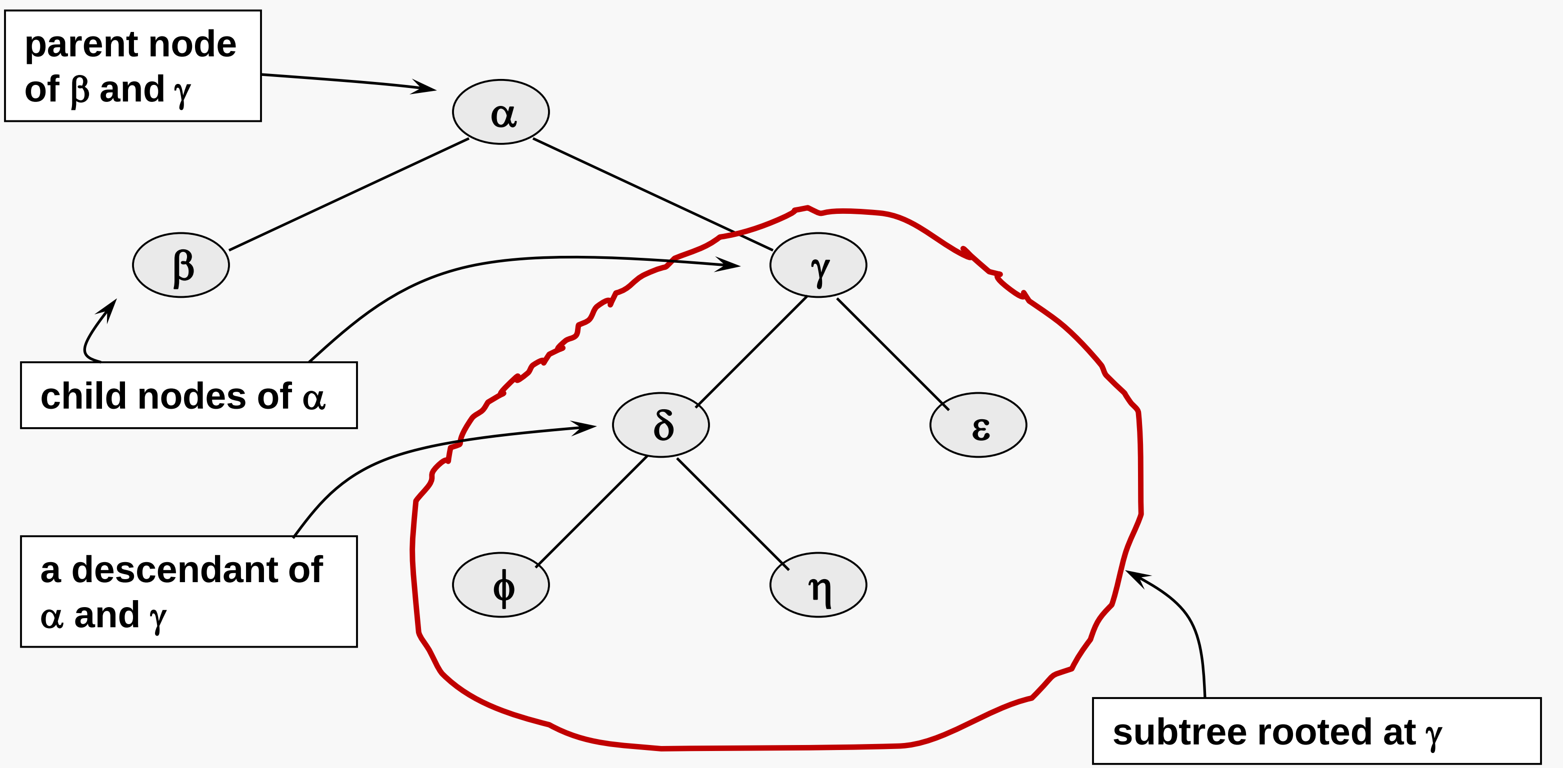


Binary Tree Node Relationships

A binary tree node may have 0, 1 or 2 child nodes.

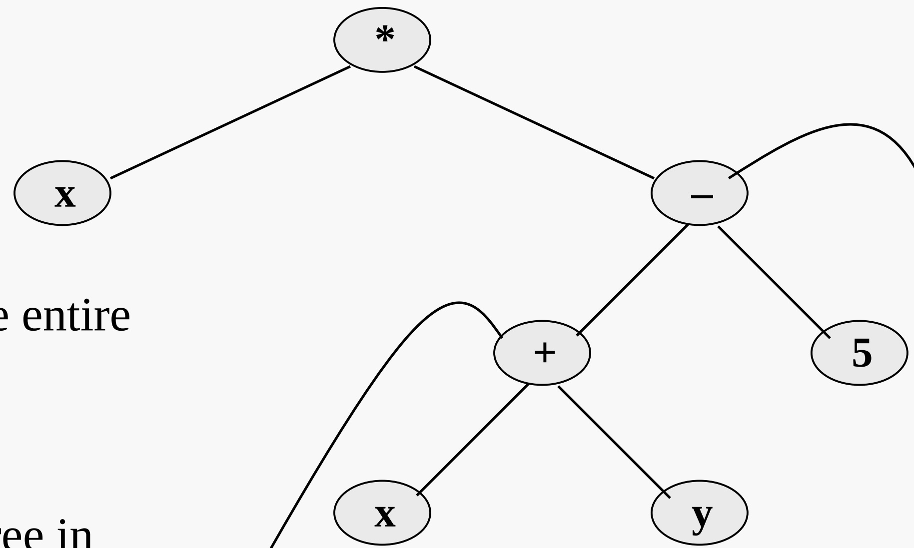
A path is a sequence of adjacent (via the edges) nodes in the tree.

A subtree of a binary tree is either empty, or consists of a node in that tree and all of its descendent nodes.



Quick Application: Expression Trees

A binary tree may be used to represent an algebraic expression:



Each subtree represents a part of the entire expression…

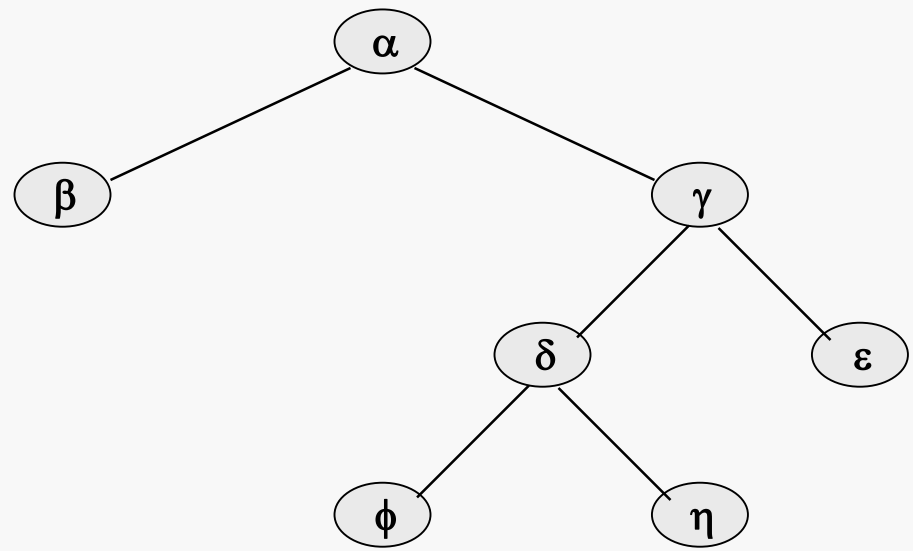
If we visit the nodes of the binary tree in the correct order, we will construct the algebraic expression:

$$x \times ((x + y) - 5)$$

Traversals

A traversal is an algorithm for visiting some or all of the nodes of a binary tree in some defined order.

A traversal that visits every node in a binary tree is called an enumeration.



preorder: visit the node, then the left subtree, then the right subtree

α β γ δ ϕ η ε

postorder: visit the left subtree, then the right subtree, and then the node

β ϕ η δ ε γ α

inorder: visit the left subtree, then the node, then the right subtree

β α ϕ δ η γ ε

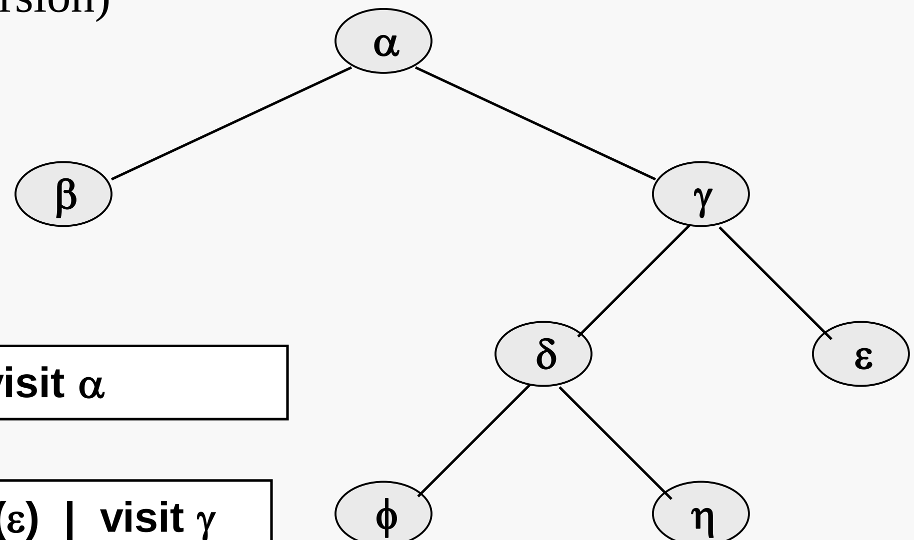
Postorder Traversal Details

Consider the postorder traversal from a recursive perspective:

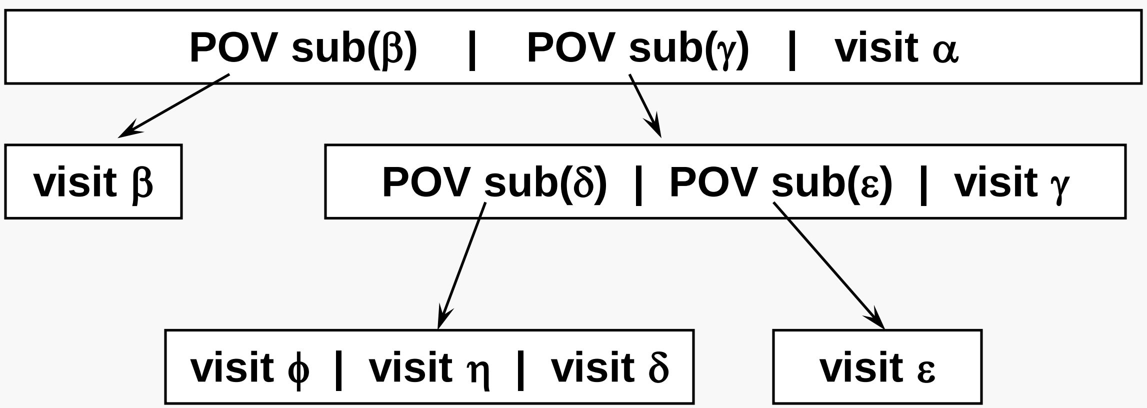
postorder: postorder visit the left subtree,

postorder visit the right subtree,

then visit the node (no recursion)



If we start at the root:



Binary Search Trees

The general binary tree shown in the previous chapter is not terribly useful in practice. The chief use of binary trees is for providing rapid access to data (indexing, if you will) and the general binary tree does not have good performance.

Suppose that we wish to store data elements that contain a number of fields, and that one of those fields is distinguished as the key upon which searches will be performed.

A binary search tree or BST is a binary tree that is either empty or in which the data element of each node has a key, and:

1. All keys in the left subtree (if there is one) are less than the key in the root node.
2. All keys in the right subtree (if there is one) are greater than (or equal to)* the key in the root node.
3. The left and right subtrees of the root are binary search trees.

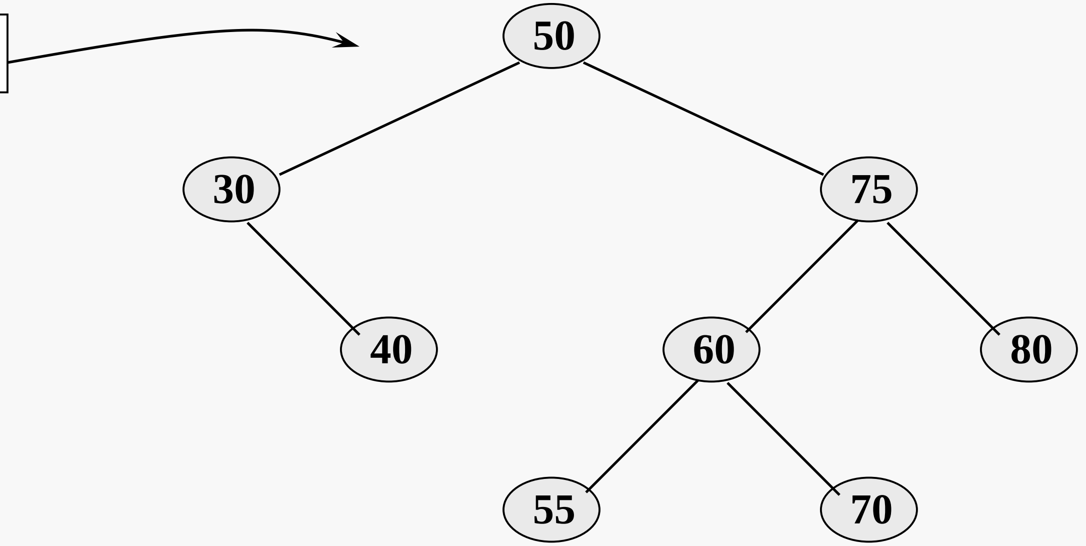
* In many uses, duplicate values are not allowed.

BST Insertion

Here, the key values are characters (and only key values are shown).

Inserting the following key values in the given order yields the given BST:

50 75 80 60 30 55 70 40



In a BST, insertion is always at the leaf level. Traverse the BST, comparing the new value to existing ones, until you find the right spot, then add a new leaf node holding that value.

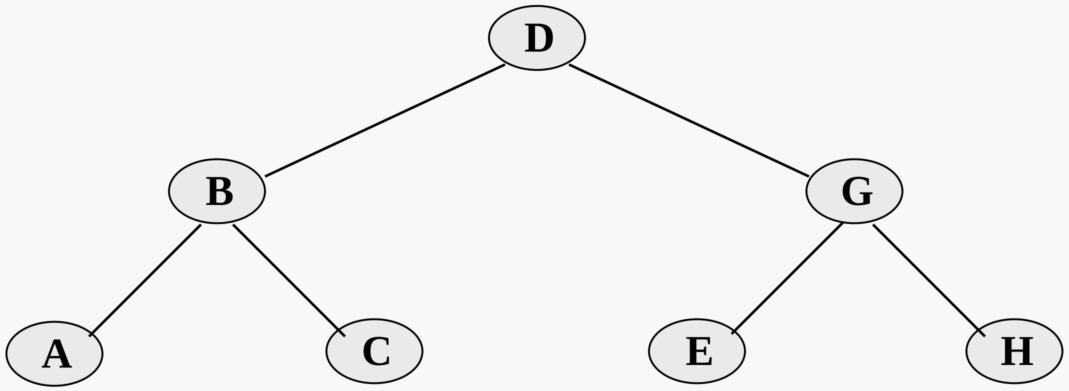
What is the resulting tree if the (same) key values are inserted in the order:

30 40 50 55 60 70 75 80

Searching in a BST

Because of the key ordering imposed by a BST, searching resembles the binary search algorithm on a sorted array, which is O(log N) for an array of N elements.

A BST offers the advantage of purely dynamic storage, no wasted array cells and no shifting of the array tail on insertion and deletion.

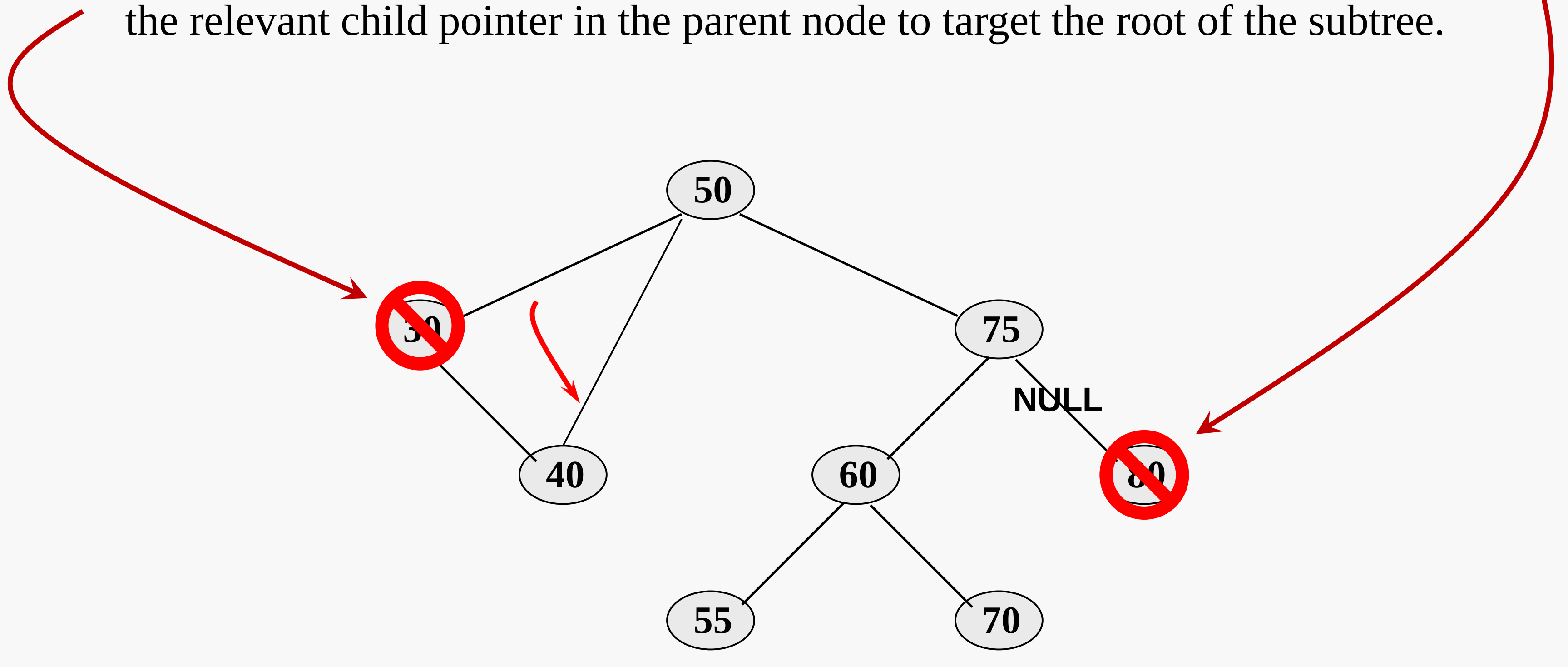


Trace searching for the key value E.

BST Deletion

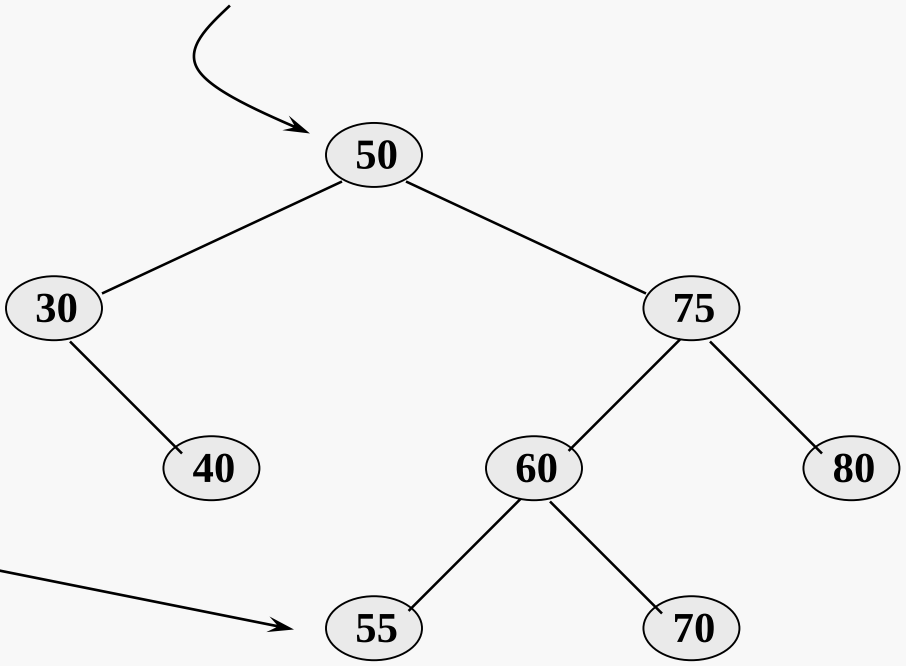
Deletion is perhaps the most complex operation on a BST, because the algorithm must result in a BST. The question is: what value should replace the one deleted? As with the general tree, we have cases:

- Removing a leaf node is trivial, just set the relevant child pointer in the parent node to NULL
- Removing an internal node which has only one subtree is also trivial, just set the relevant child pointer in the parent node to target the root of the subtree.



BST Deletion

- Removing an internal node which has two subtrees is more complex…



Simply removing the node would disconnect the tree. But what value should replace the one in the targeted node?

To preserve the BST property, we must take the smallest value from the right subtree, which would be the closest succcessor of the value being deleted

OR

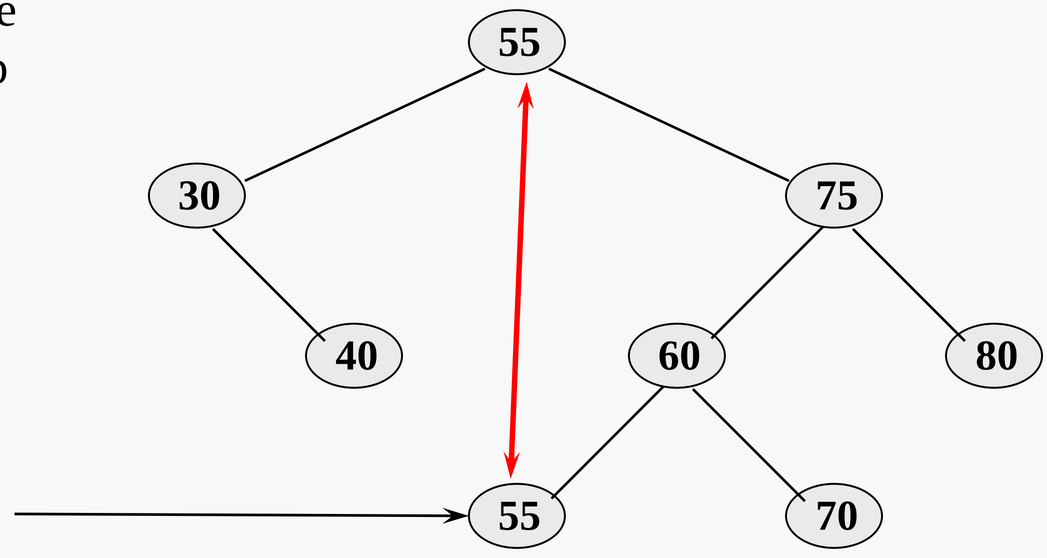
the largest value from the left subtree, which would be the closest predecessor of the value being deleted

Fortunately, these extreme values are easy to find…

So, we first find the left-most node of the right subtree, and then swap data values between it and the targeted node.

Note that at this point we don't necessarily have a BST.

Now we must delete the copied value from the right subtree.

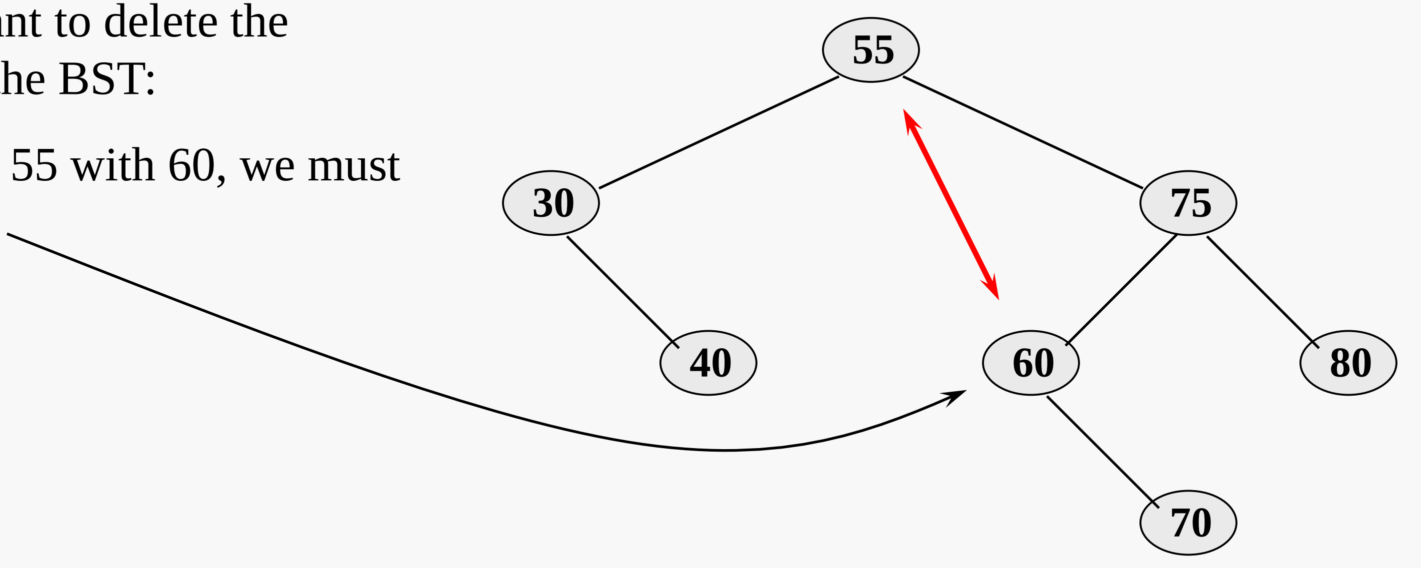


That looks straightforward here since the node in question is a leaf. However…

- the leftmost in the subtree node will NOT be a leaf in all cases
- the occurrence of duplicate values is a complicating factor
- so we might want to have a `DeleteRightMinimum()` function to clean up at this point

Suppose we want to delete the value 55 from the BST:

After replacing 55 with 60, we must delete

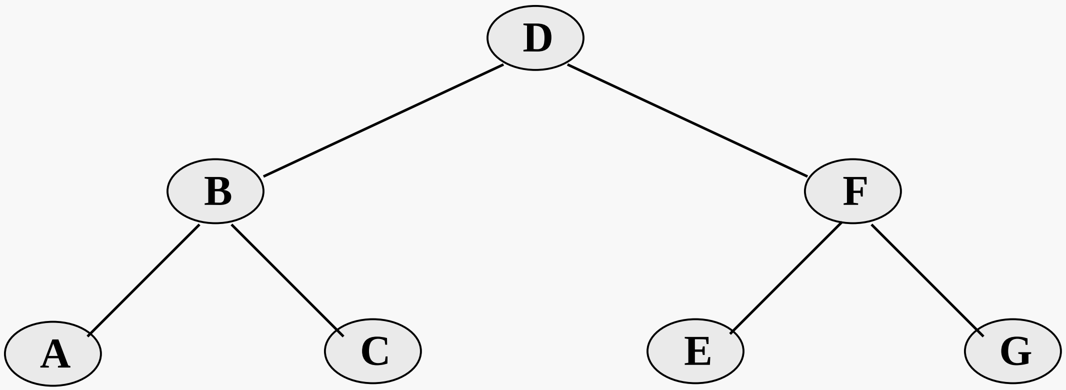


Also, consider deleting the value 75. In this case, the right subtree is just a leaf node, whose parent is the node originally targeted for deletion.

Moral: be careful to consider ALL cases when designing.

Balance in a BST

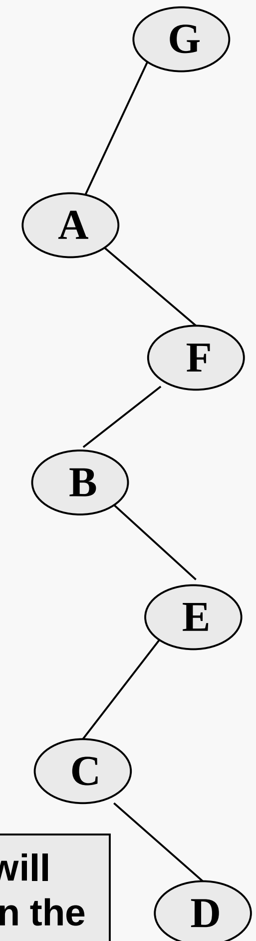
However, a BST with N nodes does not <u>always</u> provide O(log N) search times.



A well-balanced BST. This will have log(N) search times in the worst case.

What if we inserted the values in the order:

A B C D E F G



A poorly-balanced BST. This will not have log(N) search times in the worst, or even the average, case.

Search Cost in a BST

From a theorem on binary trees, we know that a binary tree that contains L nodes must contain at least 1 + log L levels.

If the tree is full, we can improve the result to imply that a full binary tree that contains N nodes must contain at least log N levels.

So, for any BST, the there is always an element whose search cost is at least log N.

Unfortunately, it can be much worse. If the BST is a "stalk" then the search cost for the last element would be N.

It all comes down to one simple issue: how close is the tree to having minimum height?

Unfortunately, if we perform lots of random insertions and deletions in a BST, there is no reason to expect that the result will have nearly-minimum height.

Cost of Insertion/Deletion in a BST

Clearly, once the parent is found, the remaining cost of inserting a new node in a BST is constant, simply allocating a new node object and setting a pointer in the parent node.

So, insertion cost is essentially the same as search cost.

For deletion, the argument is slightly more complex. Suppose the parent of the targeted node has been found.

If the targeted node has only one subtree, then the remaining cost is resetting a pointer in the parent and deallocating the node; that's constant.

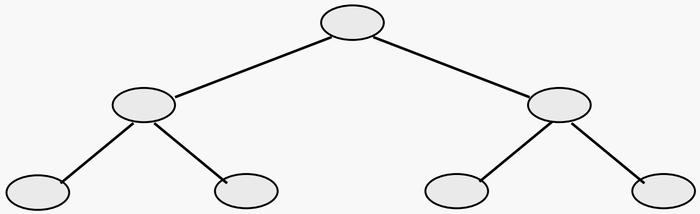
But, if the targeted node has two subtrees, then an additional search must be carried out to find the minimum value in the right subtree, and then an element copy must be performed, and then that node must be removed from the right subtree (which is again a constant cost).

In either case, we have no more than the cost of a worst-case search to the leaf level, plus some constant manipulations.

So, deletion cost is also essentially the same as search cost.

Traversal Cost in a BST

Let's suppose that we have a binary tree such that every level of the tree is completely full. For example:



Obviously, we could "extend" any given binary tree to have this property, and a tree like this lets us get a bound on the worst case for a traversal.

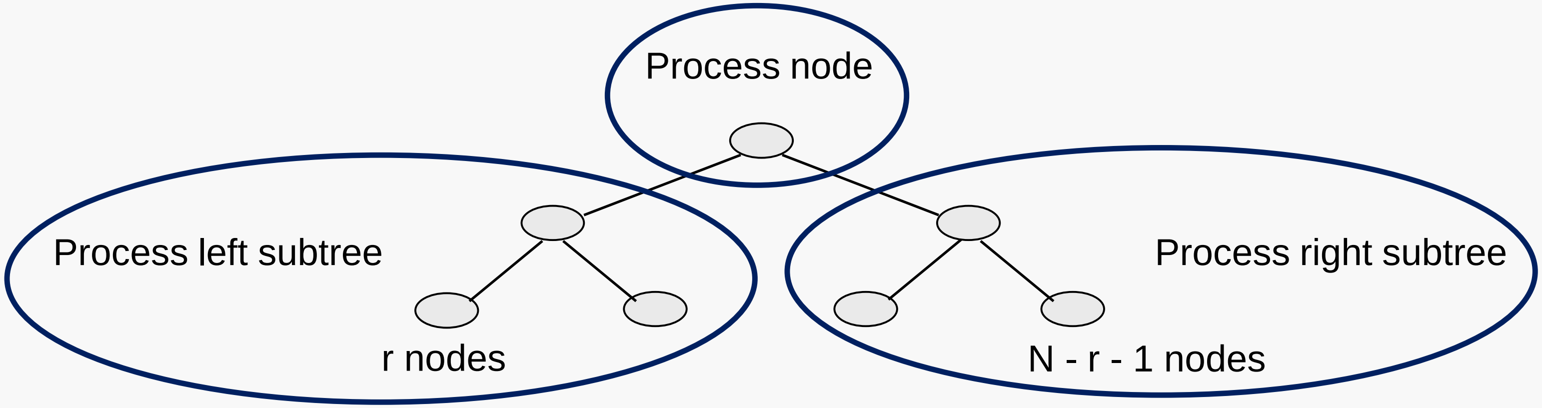
Each of the three standard traversals travels to every node in the tree, and the cost of the traversal really isn't influenced by the order in which the current node is processed.

For the sake of argument, let's say that the cost of whatever we do with a node is 1.

And, let's let $W(N)$ represent the cost of traversing a binary tree with N nodes.

Traversal Cost in a BST

Whatever node the traversal is currently at, we must do three things:



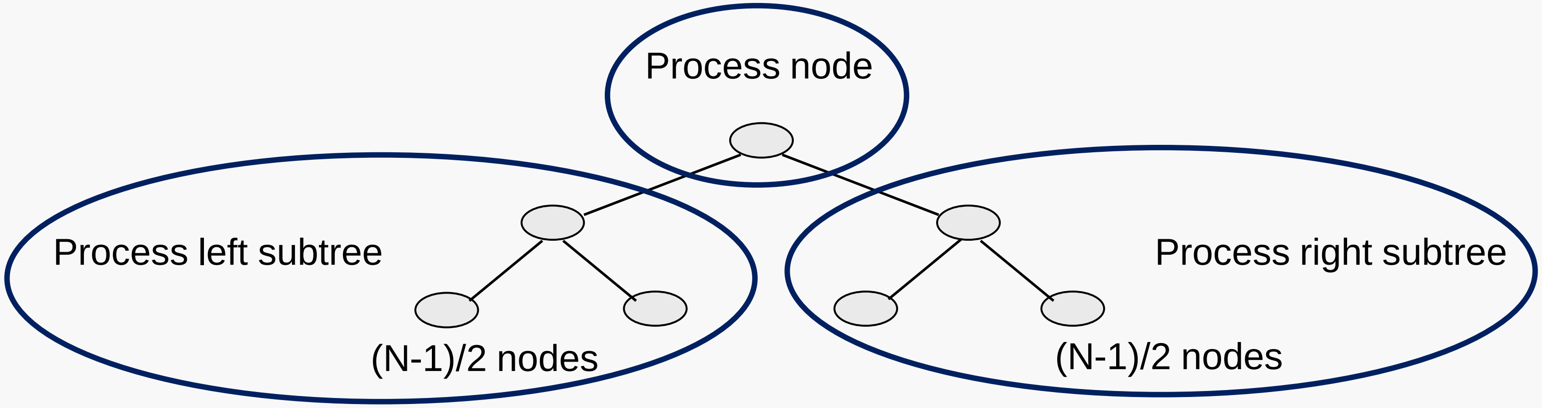
Therefore, we can write a recursive definition for the cost of the traversal:

$$W(N) = \begin{cases} N & \text{if } N = 0,1 \\ 1 + W(r) + W(N - r - 1) & \text{if } N > 1 \end{cases}$$

We will see later how to analyze this precisely…

Traversal Cost in a Full, Complete BST

In this special case, the subtrees of each node contain the same number of nodes:



This leads to a simpler recurrence for the cost of a traversal:

$$W(N) = \begin{cases} N & \text{if } N = 0,1 \\ 1 + 2W\left(\dfrac{N-1}{2}\right) & \text{if } N > 1 \end{cases}$$

This is straightforward to analyze, given one key fact…

Traversal Cost in a Full, Complete BST

If a full, complete binary tree has N nodes, then: $N = 2^k - 1$

Given that fact:

$$W(N) = W\left(2^k - 1\right) = 1 + 2W\left(\frac{2^k - 1 - 1}{2}\right) = 1 + 2W\left(2^{k-1} - 1\right)$$

Noticing the pattern above:

$$\begin{aligned}
W(N) &= W\left(2^k - 1\right) \\
&= 1 + 2W\left(2^{k-1} - 1\right) \\
&= 1 + 2\left(1 + 2W\left(2^{k-2} - 1\right)\right) \\
&= 1 + 2\left(1 + 2\left(1 + 2W\left(2^{k-3} - 1\right)\right)\right) \\
&= \cdots \\
&= 1 + 2\left(1 + 2\left(1 + \cdots + 2W\left(2^1 - 1\right)\right)\right) \\
&= 1 + 2 + 2^2 + 2^3 + \cdots + 2^{k-1} \\
&= 2^k - 1 \\
&= N
\end{aligned}$$

So, for a full, complete binary tree, a full traversal is *O*(N).

Note:

This is NOT a proof by Induction.

This is an intuitive argument by vigorous hand-waving.