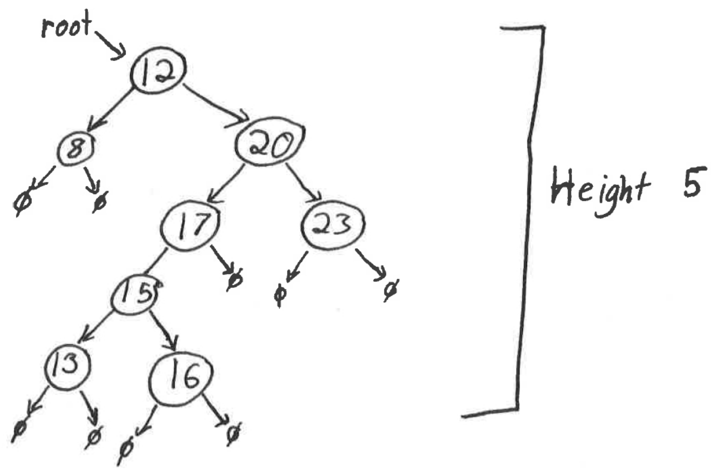
A **Binary Search Tree (BST)** is a hierarchical datastructure, every node in the tree has two pointers, a **left** child and a **right** child pointer.

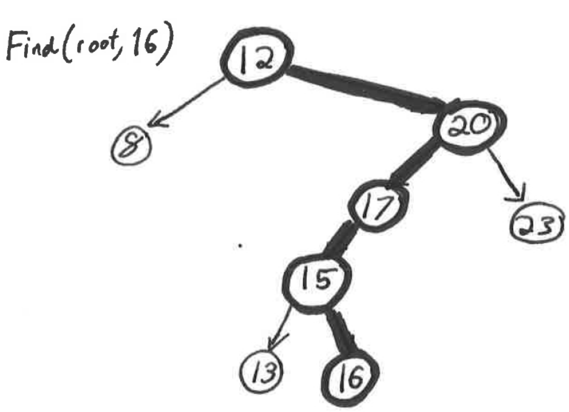
For any node in the tree, all of the nodes <u>to the left</u> of the given node <u>are lesser in value</u>, all of the nodes <u>to the right</u> are <u>greater in value.</u> Each child is unique (two nodes <u>cannot</u> point to the same child). Every child pointer must point to either a unique node or point to NULL. The following is an example of a BST.




Note: the null pointers are shown for completeness. Sometimes we do not show the null pointers in order to save space. The top of the tree is called the **root**. The **height** of the tree is the number of non-null nodes from the root to the bottommost leaf. The **depth** is the distance from the root to the current node. The root is always of depth 0, nodes 8 and 23 (in the diagram) are of depth 1. Nodes 17 and 23 have a depth 2, etc. All nodes have depth less than the height of the tree.

A binary search tree is so called, because we are allowed to use the "binary search" algorithm in order to find a node in the tree. We implement a "Find" algorithm by starting at the root, and performing binary search until we find the node of interest. In the following example we find the node "16".

Compare 16, 12,    16 is greater than 12 recurse right
Compare 16, 20,    16 is less than 20 recurse left
Compare 16, 17,    16 is less than 17 recurse left
Compare 16, 15,    16 is greater than 15 recurse right
Compare 16, 16,    16 is equal to 16, so we found the node!   Return x

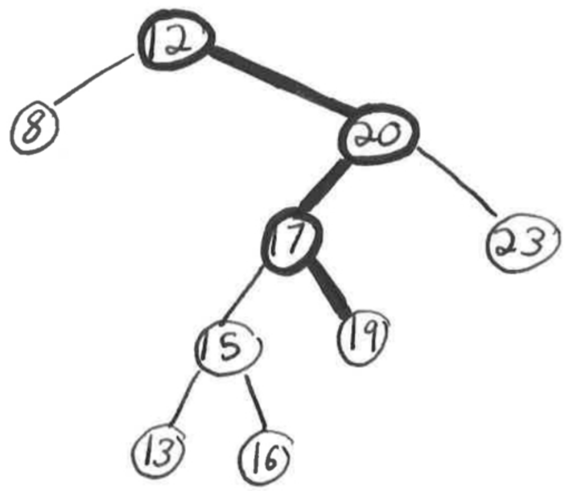



```
Find (x, val)
{
   if (x == Null)
       return Null
   if (x -> val == val)
       return x
   if (val < x -> val)
       return Find (x -> L, val)
   else
       return Find (x -> R, val)
}
```

To Insert a node, it is similar to find, we need to traverse the tree from the root down until we find a NULL pointer, and then insert the new node at that pointer. In this example we

Compare 19 to 12, and 19>12, insert right.
Compare 19 to 20 19<20 insert left.
Compare 19 to 17 19>17 insert right but right is NULL so insert right there!

Insert (root, 19)




We have to be careful to keep track of the parent pointers when we insert, that's why the Pseudocode for insert differs slightly from that of Find. However, in principle, the two operations work in similar ways.

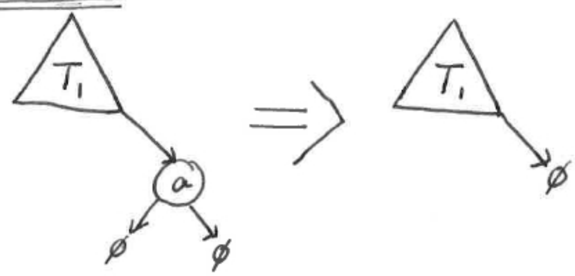
```
Insert (x, val)
{
    if (x = Null)
        root = new Node(val)
    if (val < x -> val)
    {
        if (x -> L = Null)
            x -> L = new Node (val)
        else
            Insert (x -> L, val)
    }
    else if (x -> val < val)
    {
        if (x -> R = Null)
            x -> R = new Node (val)
        else
            Insert (x -> R, val)
    }
}
```

The Remove operation is a bit more complicated than Insert or Find, because Remove has two cases. Either the node to remove ‘a’ has zero children or one children, in which case we can remove the node without causing any problems with the subsequent nodes.

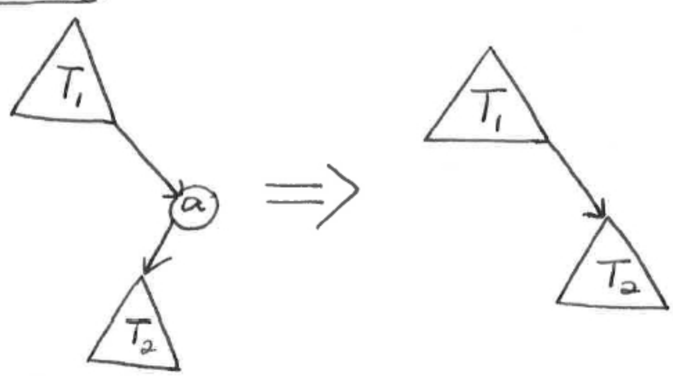
However, there is a more challenging case in the event that the node ‘a’ has <u>two children</u>. In this case we cannot just delete ‘a’ because then both of it’s children would be dangling pointers.

Instead, we <u>replace the value</u> of a with the <u>value of it’s Successor</u>, and then remove the successor. The successor is the very next largest node in sorted order of the binary search tree.

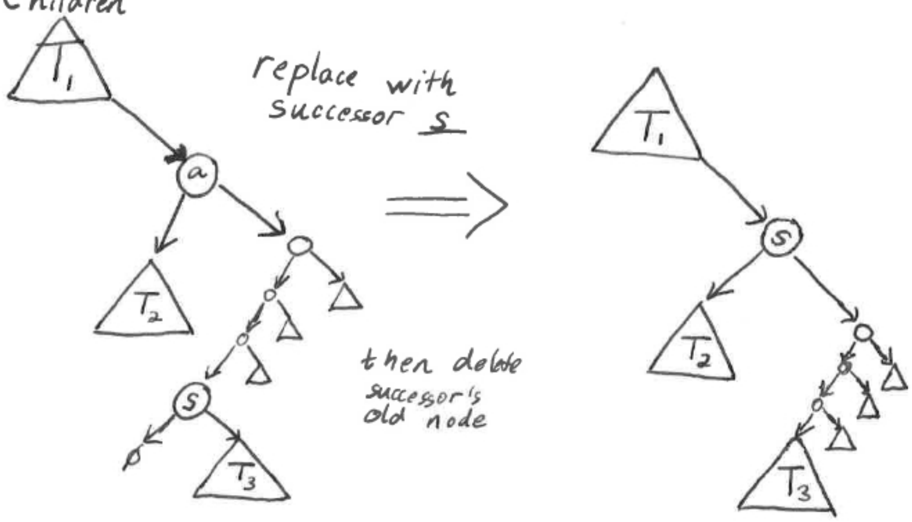
No Children




One Child




Two Children



We can see psuedocode for “Remove” as calling the “FindSuccessor” function in the event that we need to remove a node with two children.

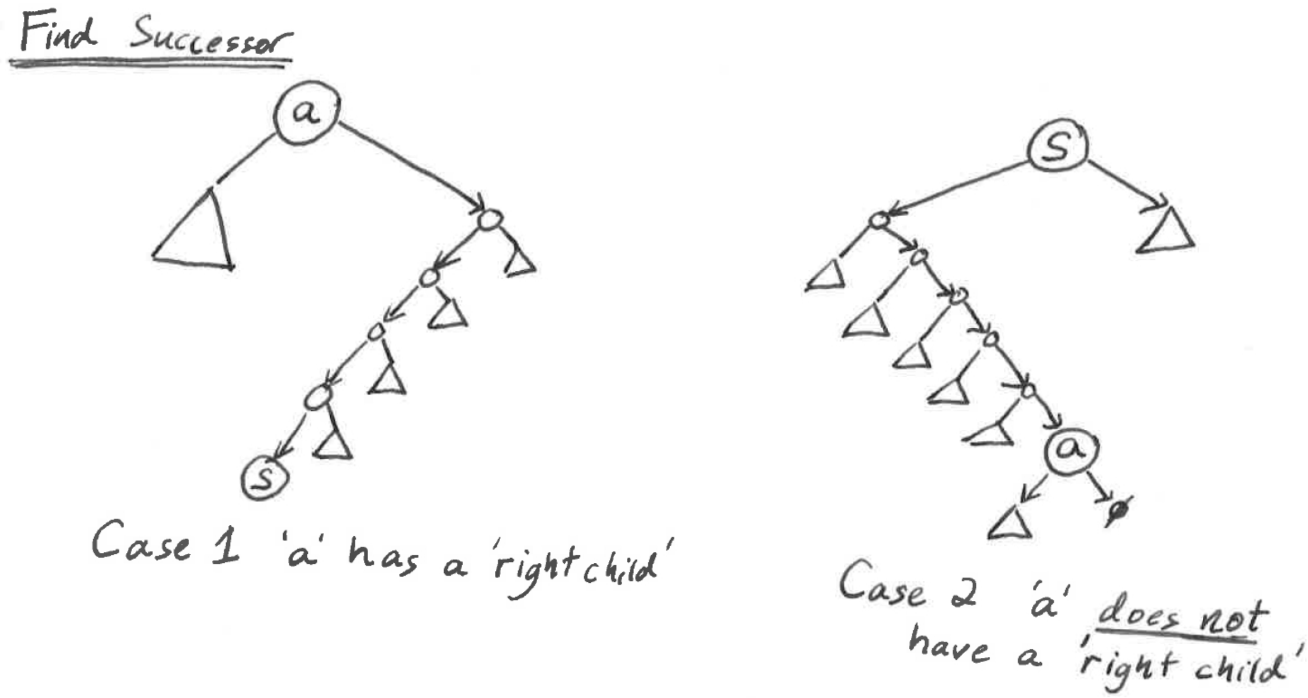
```
Remove (x, parent, val)
{
    if (x == Null)
        return
    if (val < x -> val)
        Remove (x -> L, x, val)
    else if (x -> val < val)
        Remove (x -> R, x, val)
    else
    {
        // we've found the node to remove
        if (x -> L == Null)
        {
            parent = x -> R
            delete x
        }
        else if (x -> R == Null)
        {
            parent = x -> L
            delete x
        }
        else
        {
            s = Find Successor (x)
            x -> val = s -> val
            Remove (s, val)
        }
    }
}
```

The <u>Successor</u> is the next element of the tree in <u>sorted order</u>. However, sometimes the successor is a descendant of the node (one of it’s children’s children’s children), and sometimes it is an ancestor.

Case 1: If the node has a right child, then the successor is within the right branch of the descendents. All of it’s right children are larger in value than the node, and the *“leftmost descendent of the right child”* is the successor if the node has a right child.

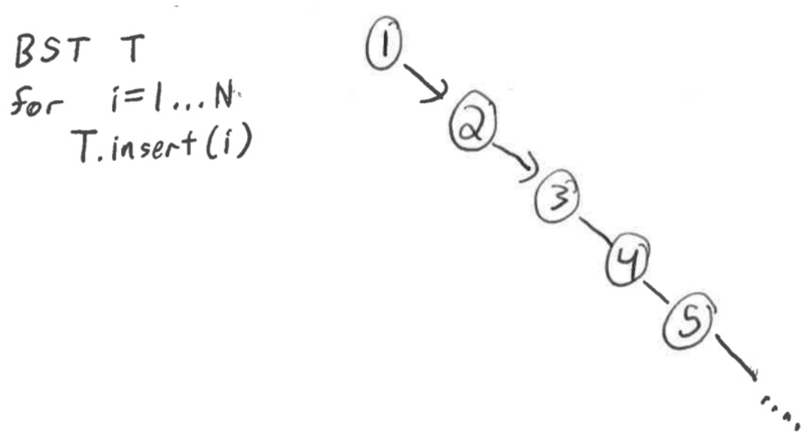
Case 2: If the node does not have a right child, then ‘a’ is the predecessor of it’s successor, so *“a is the rightmost descendent of it’s successors left child”*

These cases of FindSuccessor are illustrated in the following diagram.




Typically all of the operations (Insert, Find, and Remove) take O lg N time for "randomly inserted trees". However, the true amount of time for each of these operations is O H (big Oh of the Height).

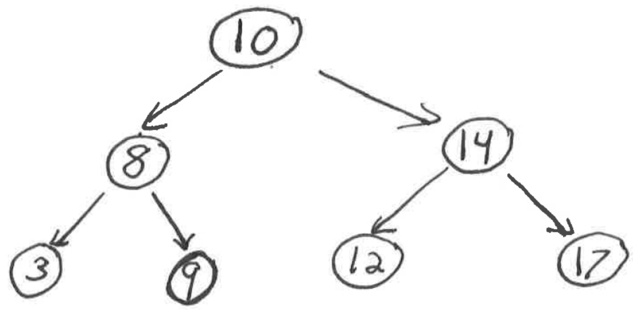
Ordinary Binary Search Trees <u>do not have any guarantee</u> that H is O lg N. Even though it usually is, it is very easy to write a for loop that inserts the nodes in an order such that H = N, and the binary tree becomes a linked list.




Note: that AVL trees have a guarantee that H is order lg N, and the Induction notes prove this fact! However, ordinary BST do not guarantee this.

Regardless of the shape of the tree, one thing that always takes <u>linear time</u> is traversal to print out all of the nodes of a tree. We describe three traversals:

PreOrder traversal, InOrder traversal, and PostOrder traversal




Preorder: 10 8 3 9 14 12 17

In order: 3 8 9 10 12 14 17

Post order: 3 9 8 12 17 14 10

**InOrder** traversal is <u>often used</u> to print out the nodes in sorted order.
**PostOrder** traversal is <u>often used</u> to delete the nodes because you must delete the children first before ou delete yourself (unless you want memory leaks, valgrind errors and segmentation faults.

```
In Order (x)
{
    if (x = Null)
        return
    In Order (x -> L)
    print x
    In Order (x -> R)
}
```

```
Pre Order (x)
{
    if (x = Null)
        return
    print x
    Pre Order (x -> L)
    Pre Order (x -> R)
}
```

```
Post Order (x)
{
    if (x = Null)
        return
    Post Order (x -> L)
    Post Order (x -> R)
    delete x
}
```

Note that all tree traversals take <u>linear time</u> of number of nodes O N
whereas Insert, Find, and Remove take time proportional to height O H