# Inserting a Node into a Binary Tree

**As you might imagine, inserting a node into a binary search tree is somewhat similar to searching for a node. In some sense, you "trace out" the same path. Thus, if we are to insert a node recursively, here is our basic strategy:**

**1) If the tree is empty, just return a pointer to a node containing the new value.**

**2) Otherwise, see which subtree the node should be inserted by comparing the value to insert with the value stored at the root.**

**3) Based on this comparison, recursively either insert into the left subtree, or into the right subtree.**

**This basic plan is just fine, but we'll have to slightly modify it to account for a couple details:**

**Just like the linked list code, we will be returning a pointer to the root of the tree. (This is necessary in the cases where the root of the tree changes. If we don't do this, we must pass in a double pointer.)**

**We should not attempt to directly insert into an "empty tree" unless the initial tree is empty. This may sound strange, but essentially, we don't want to "lose" our link to the tree through a recursive call on a NULL node.**

**To do this, once we decide to go right or left, we have one more decision to make:**

**Is that link NULL? If so, attach the node w/o a recursive call.**
**If not, make the recursive call.**

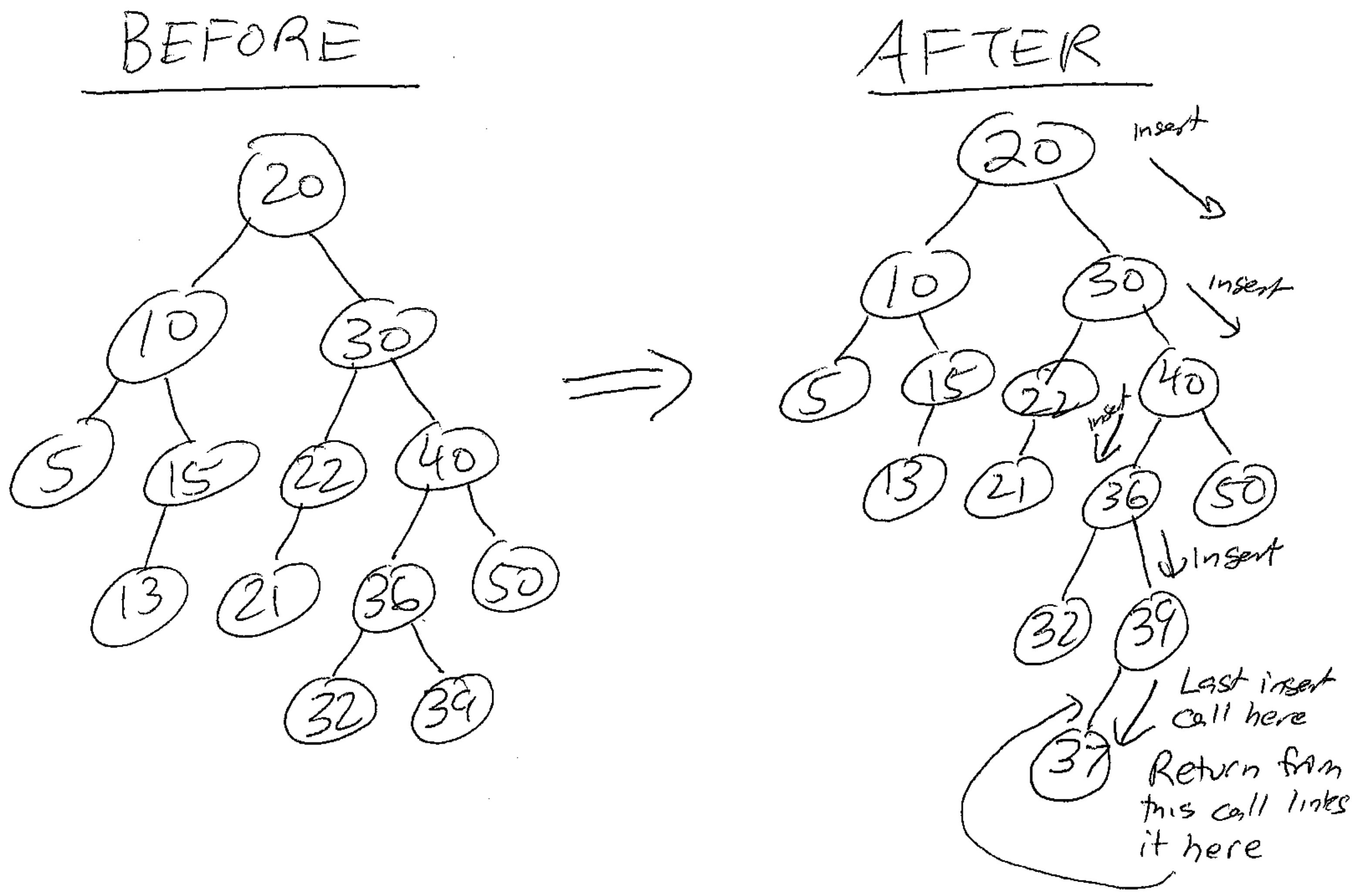
**Here is the corresponding recursive insert code:**

```
tree_node* insert(tree_node *current_ptr, int val) {

  if (current_ptr == NULL) {
    tree_node* tmp = malloc(sizeof(tree_node));
    tmp->data = val;
    tmp->left = NULL;
    tmp->right = NULL;
    return tmp;
  }

  if (val <= current_ptr->data)
    current_ptr->left = insert(current_ptr->left, val);
  else
    current_ptr->right = insert(current_ptr->left, val);
                                              typo: current_ptr->right
  return current_ptr;
}
```

**Here is picture of inserting the value 37 into the tree shown below to the left:**



## Summing the nodes in a binary tree

**We can really use any of the traversals to implement this. All we need to do add the values from the three portions of the three together and return this answer. Notice how succinct this code is!**

```
int add(tree_node *current_ptr) {

  if (current_ptr == NULL) return 0;

  return current_ptr->data+Add(current_ptr->left)+
                            Add(current_prt->right);
}
```

## Search of an arbitrary binary tree

**We have already looked at searching for a node in a binary search tree. Now consider the problem if the tree is NOT a binary search tree. This time, we must recursively search both subtrees after looking at the root node. (Also, a traversal could be very easily adapted for this task!)**

```
int search(tree_node *current_ptr, int val) {

  if (current_ptr == NULL)      return 0;
  if (current_ptr->data == val) return 1;

  return search(current_ptr->left, val) ||
         search(current_ptr->right, val));
}
```

## Height of an arbitrary binary tree

**The height of a binary tree is defined as the longest path from the root to any leaf node. An empty tree has height -1 and a tree with 1 node has height 0. When we view it recursively, we want the heights of both the left and right subtrees, and between those, we only care about the taller one.**

```
int height(tree_node *current_ptr) {

  if (current_ptr == NULL)      return -1;

  int leftH = height(current_ptr->left);
  int rightH = height(current_ptr->right);

  if (leftH > rightH)
    return 1 + leftH;

  return 1 + rightH;
}
```

## Class Exercise

**Write a function that prints out all the values in a binary tree that are greater than or equal to a value passed to the function. (The tree may not be a binary search tree.) Here is the prototype:**

**void PrintBig(tree_node *current_ptr, int value);**