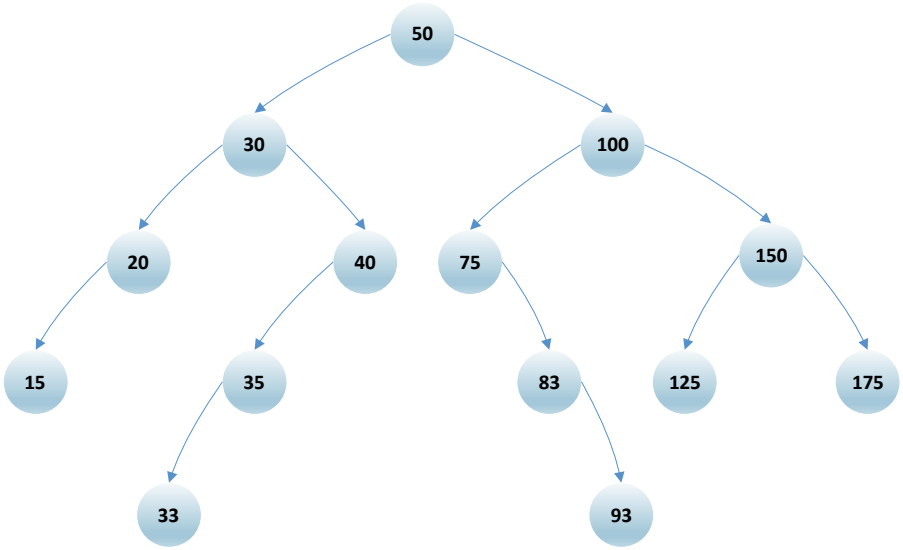
PLAYBOOK NOTES: BINARY TREE – DELETING A NODE

There are four cases when deleting nodes:

1- Deleting a leaf node
2- Deleting a node with only a left child
3- Deleting a node with only a right child
4- Deleting a node with two children

Before we can delete a node, we will need to find the location of the node that we want to delete. To accomplish this, we will call a function called SearchTree.

Four examples, we will assume the following tree has been established:



Writing the SearchTree function:

There are two pointers that are needed to delete a node. One pointer will be a pointer to the **parent** node of the node that we want to delete and a **current** pointer that will point to the node that we want to delete. This is like a link list when we used the current and previous pointer.

Return Type:

- This function will return a true/false value based on if the value you are searching for has been found.

Parameters:

- The value to search for – In this example, this parameter would be an integer
- A pointer to your root node – This parameter will not change in this function
- A pointer called pCurrent – This parameter will be updated in this function
- A pointer called pParent – This parameter will be updated in this function

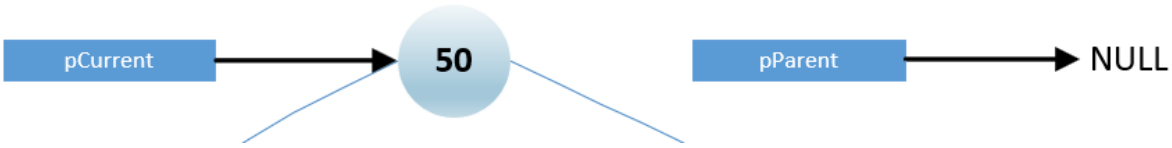
Algorithm:

Start the function by creating a return value and assuming that the result is not found, this variable will be assigned false or 0.

Check to see if the **root** node is NULL, if it is, the **current** and **parent** pointer will be assigned NULL.

If there is at least one node in the tree, assign the **current** pointer equal to the **root** pointer and assign the **parent** pointer to NULL.

Assuming the tree above, our pointers will now look like this:



We now enter a while loop that will continue to loop while the **current** pointer does not equal NULL and the value to search is not found.

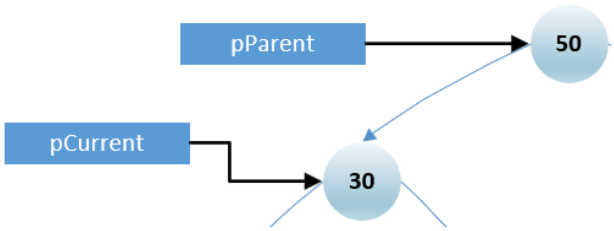
Inside the while loop:

If the value of the **current** node is equal to the value to search for, set the return variable to true.

If the previous if-statement fails, then proceed to the next if-statement.

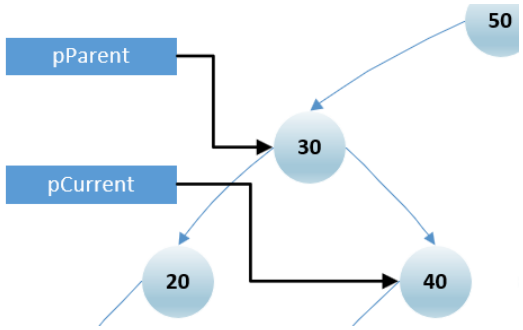
If your **search value** is less than the value pointed to by the **current** node, assign the **parent** pointer to the **current** pointer and move the **current** pointer to the left.

Because we are searching for 40, this if statement would be true and our pointer will now look like this:



If the previous if-statement fails, then we move to the right. So, we assign the **parent** pointer to the **current** pointer and move the **current** pointer to the right.

Because we are searching 40, we will fail both if-statement on the second pass through the while loop. After moving to the right one spot our pointers look like this:



The next time through the while-loop, the first if-statement will be true, and the return value will be set to true.

After the while loop is done executing, return your return value.

Function: DeleteNode

Return Type:

- This function has no return type

Parameters:

- A pointer your root node – This parameter could change in this function
- An integer value representing the node you want to delete

Algorithm:

You will want to declare your **current** and **parent** pointers.

This will be the function where you call the SearchTree function to initialize these pointers.

If your value to delete was found then based on all the pointers, you will use a series of if-statements to call the functions above.

HINT: All your if-statements test will be on the **current** pointer.

Writing the function DeleteLeafNode:

Return Type:

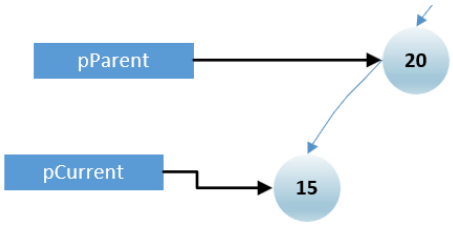
- This function has no return type.

Parameters:

- A pointer to your root node – This parameter could be updated in this function
- A pointer called pCurrent – This parameter could be updated in this function
- A pointer called pParent – This parameter could be updated in this function

Algorithm:

In this function we will delete a leaf node. *Using the tree above, we will assume that the node we want to delete is 15. Our pointers from SearchTree will look like this:*



Check to make sure that **parent** pointer does not equal NULL. If we have a valid pointer, then proceed. If the **parent** pointer is NULL, then (this is the last node) set the **root** pointer to NULL and free the **current** pointer.

If the **parent** pointer is not equal to NULL, check to see if the node pointed to by the **left link of the parent** is equal to the **current** pointer. If this is true, then set the **left link of the parent** pointer to NULL and then free the **current** node.

In the example above, this if-statement is true, and this will remove 15 from the tree.

If the previous if-statement is false, then we can assume that the **right link of the parent** is equal to the **current** pointer. So, set the **right link of the parent** pointer to NULL and then free the **current** node.

Writing the function DeleteNodeWithLeftChildOnly:

Return Type:

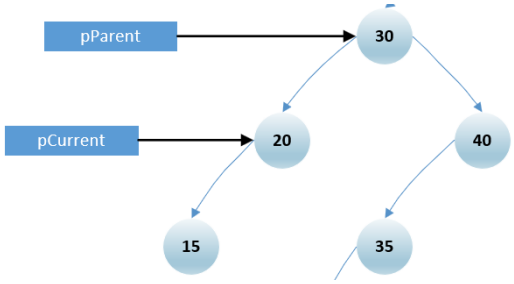
- This function has no return type.

Parameters:

- A pointer to your root node – This parameter could change in this function
- A pointer called pCurrent – This parameter could be updated in this function
- A pointer called pParent – This parameter could be updated in this function

Algorithm:

For this function we will assume the 15 is still in the tree and that the node we wish to delete is the 20. After calling SearchTree, our pointers will look like this:



It may be worth noting that there could be more nodes below 15. If this is the case, it would not change the following algorithm.

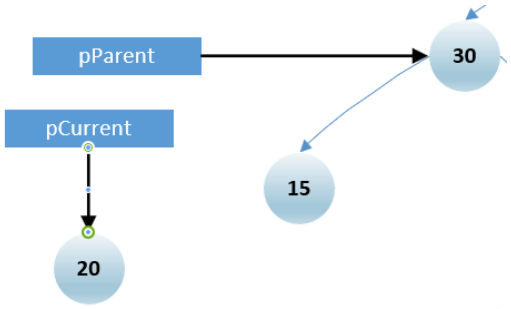
Check to make sure that the **parent** pointer does not equal NULL. If this test passes, then do the following:

Check to see if the **current** pointer is equal to the node pointed to by the **parent's right link**. If this test passes, re-assign the **parent's right link** to equal the node pointed to by the **currents left link**. Then free the **current** node.

Using the example above, this if-statement would fail.

If the previous if-statement fails, then we can assume that the **current** pointer is equal to **parent's left link**. So, re-assign the **parent's left link** to equal the node pointed to by the **currents left link**. Then free the **current** node.

Using the example above, the else portion of the if-statement would execute. Re-assigning the parent's left link to equal the currents left node would make the tree look like this: (You still must free the node pointed by the current pointer in the picture below)



If the first if-statement fails (and the **parent** pointer was passed in as NULL), then the current node must equal the root node and we know there are no other nodes on the right. Assign the **root** pointer to the **left link of the root** pointer (this will move the root to the left one node). Then free the **current** node.

Function: DeleteNodeWithRightChildOnly

This function is much like the previous, with a few changes that you are left to figure out.

Writing the function DeleteNodeWithTwoChildren:

Return Type:

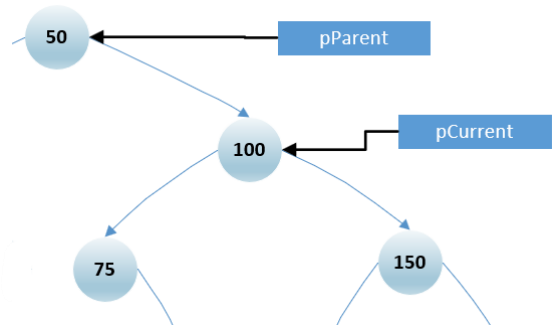
- This function has no return type

Parameters:

- A pointer to your root node – This parameter could change in this function
- A pointer called pCurrent – This parameter could be updated in this function
- A pointer called pParent – This parameter could be updated in this function

Algorithm:

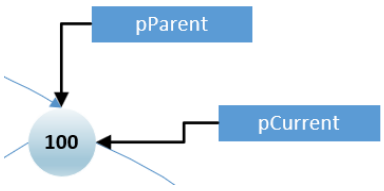
For our example, we will delete the node with value of 100. After calling SearchTree, our pointers look like this:



First create **temp** pointer that is the same data type as the **current** and **parent** pointers, except that it is a single pointer.

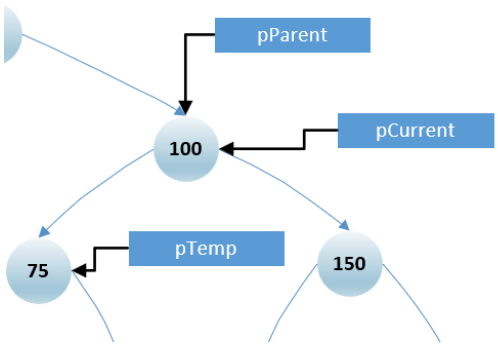
Set your **parent** pointer to equal the **current** pointer.

For our example, the pointers will look like this:



Assign the **temp** pointer to the node pointed to by the **left link of the current** pointer.

For our example, the pointers will look like this:

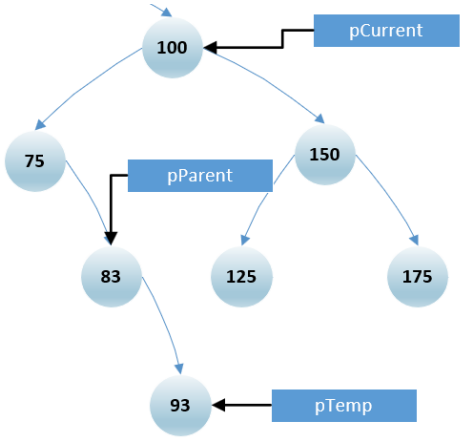


While the **right link of your temp** pointer does not equal NULL:

- Assign the **parent** pointer to the **temp** pointer
- Assign the **temp** pointer to the **right link of the temp** pointer

(We are taking the temp pointer, moving one to the left then going as far as we can to the right)

After the while loops is done executing, the pointers for our example would look like this:



After the loop is done executing, if your **parent** pointer is equal to your **current** pointer (your parent **pointer** did not move), then assign the **left link of your parent** pointer to the **left link of your temp** pointer.

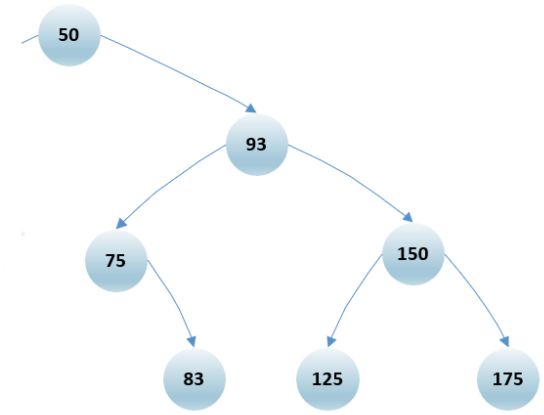
This if-statement will fail for our example.

If the previous if-statement fails, then assign the **right link of the parent** pointer to the **left link of the temp** pointer.

In our example, the left link of the temp pointer is NULL. So, the right link of the parent will be NULL and 93 will no longer be attached to the tree.

The following line of code will execute for both the true and false case of the previous if-statement:
Assign the data of the node pointed to by **current** to the data pointed by the **temp** pointer.

In our example, the 93 will take the place of the 100. Now the tree looks like this:



The 100 is gone and the all the properties of a binary search tree are intact.

Last step, free the **temp** pointer.
