Binary search Tree

Rules:

Left Child — the lesser value than the root node (which is less than any of the Right nodes).

Right Child - bigger value than the parent

* BST are beneficial because finding a node is fast log(n) since you can keep cutting the search in half

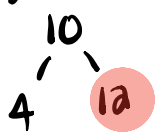
Example) put this array in a BST

[10, 4, 12, 3, 2, 5, 6, 7, 8, 9]

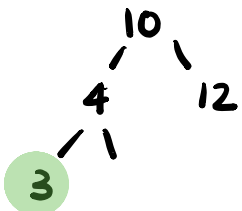
Step 1) 10 is the first element so automatically becomes the root.

Step 2) 4 is less than 10 so it is a left child

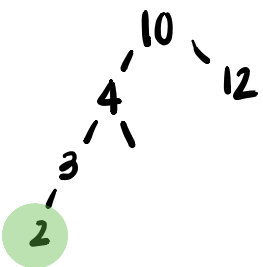
Step 3) 12 is larger than 10 so becomes a Right child



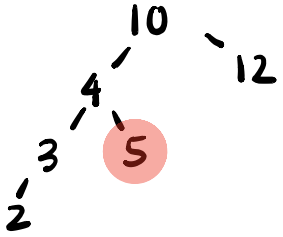
Step4) 3 is less than 10 and less than 4 so becomes a left child to 4.



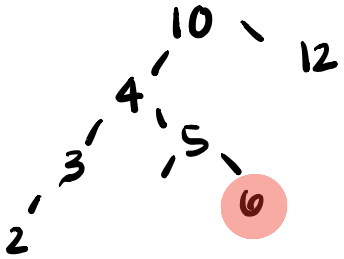
Step 5) 2 is less than 10, less than 4, less than 3 so 2 is a left child of 3.



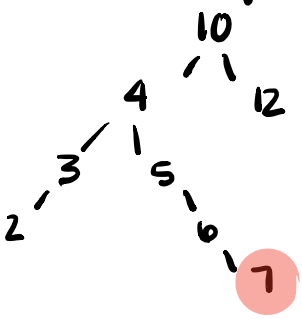
Step 6) 5 is less than 10 and greater than 4 so 5 becomes a Right Child of 4



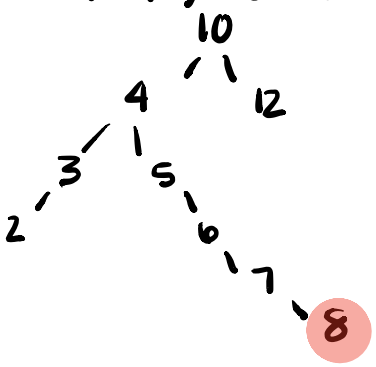
Step 7) 6 is less than 10, and greater than 4 and greater than 5 so becomes a Right Child of 5



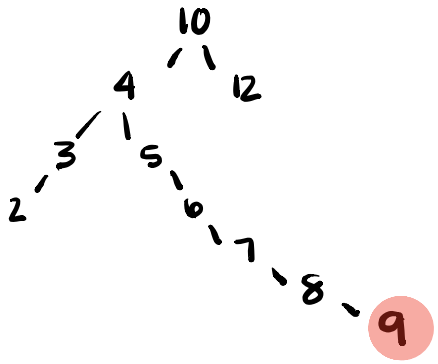
Step 8) 7 is less than 10, greater than 4, 5, and 6 so Right Child to 6.



Step 9) 8 is less than 10, greater than 4, 5, 6, 7 so R.C.

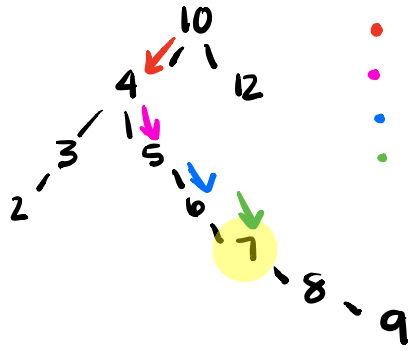


Step 10) 9 is less than 10, greater than 4, 5, 6, 7, 8



Final tree

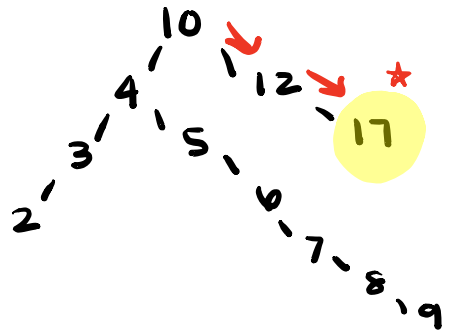
Example: TRY to find 7 is this tree



- 7 less than 10, split left
- 7 greater than 4, split Right
- 7 greater than 5, split Right
- 7 greater than 6, split Right

★ found 7! ★

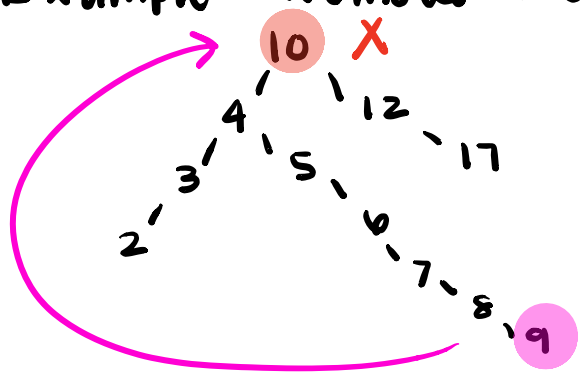
Example: Insert 17 (see algorithm)



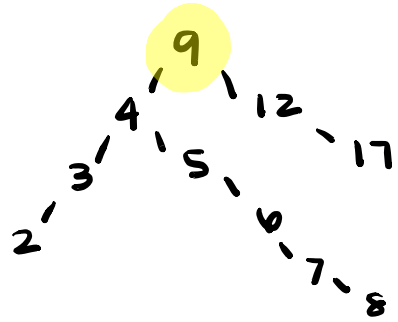
Deletion:

ROOT - Remove 5 and traverse down the left side of the tree and pick the largest value there. (so in the example above this would be 9)

Example: Remove root (10)



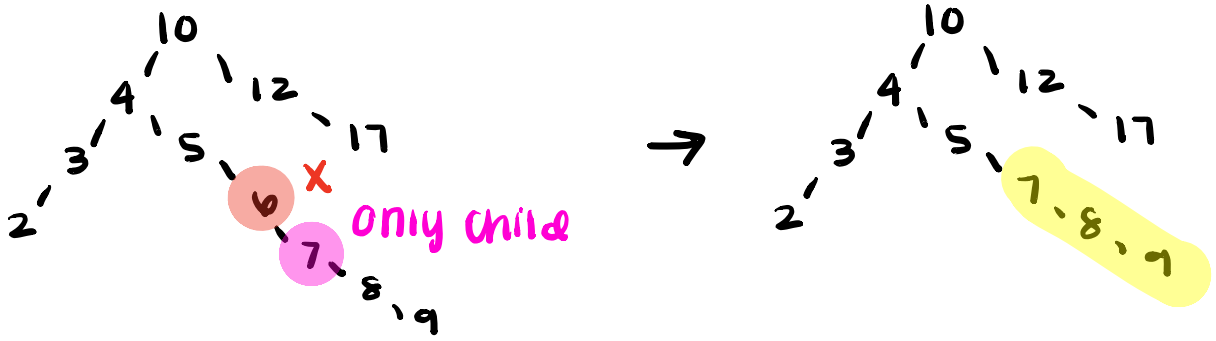
→



& and as you can see, this is still a valid BST.

Node w/ one child - traverse to the node you want to delete and the replace it with its child.

Example: delete 6



Node w/ multiple children - traverse to the node, then go to the left and replace it

Searching :

node, key value

```
def Tree-search(x, k):
    if x is None OR k is x.key:
        return x

    if k < x.key
        return Tree-search(x.left, k)

    else (k)
        return Tree-search(x.right, k)
```

IF there is no node OR the CURRENT node x you are looking at holds that value, return that x node.

value k is smaller than the value in node x so you need to go left.

the value (k) is bigger than what is in node x so look Right

Inserting :

tree, node to insert

```
def tree-insert(T, z):
    y = None
    x = T.root
    while x is not None:
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = x.right
    z.parent = y
    if y == None:
        T.root = z
    else if z.key < y.key:
        y.left = z
    else:
        y.right = z
```

While the tree, T, exists

y now = ROOT

is value at node z is less than root value (x)

then go to the left

IF value at z is greater than root value (x)

then go to the right

Changing y to be the parent value

IF there is no parent to z then z becomes the root of the tree

IF z value is less than parent value

z is a left child of its parent y

is z value is bigger than parent, y, value

z is a right child of parent y

deleting :

tree, node

```
def tree-delete(T, z):
    if z.left is None:
        transplant(T, z, z.right)
    else if z.right is None:
        transplant(T, z, z.left)
    else:
        y = tree-minimum(z.right)
        if y.parent is not z:
            transplant(T, y, y.right)
            y.right = z.right
            y.right.parent = y
        transplant(T, z, y)
        y.left = z.left
        y.left.parent = y
```

IF there is no left child then transplant with the only other child (right)

IF there is no right child then transplant with the only other child (left)

IF there are two children, then

Find the min value on the right side of the tree

transplant (used w/ delete):

```
def transplant (T, u, v):
    if u.parent is None:
        T.root = v
    else if u is u.parent.left:
        u.parent.left = v
    else u.parent.right = v
    if v is not None:
        v.parent = u.parent
```

```
def insert(self, data):

    new_node = Node(data)  # This is the node we need to insert so we create an
    # instance of a node.

    if (self.__root == None): # If there is no root, then it becomes the root.
        self.__root = new_node
    else:
        current_node = self.__root # Since there is a root, we are going to start there,
        # and move down to find where we can insert the new_node.

        parent = None  # Keeping track of the parent of current node (so initially that is root
        # but this will change)

        while(current_node != None): # Wanting to find the location/position
        # where we can insert the new_node so we are looping through until the current position
        # is not null.
            current_node.setParent(current_node)
            parent = current_node

            if data < current_node.getData():
                current_node = current_node.getLeftChild()
            else:
                current_node = current_node.getRightChild()

        # We found the right position, and now we need to insert that new_node into
        # the correct position.
        if data < parent.getData():
            new_node.setParent(parent)
            current_node = parent
            current_node.setLeftChild(new_node)

        else:
            new_node.setParent(parent)
            current_node = parent
            current_node.setRightChild(new_node)
```

```
def delete(self, data):
    # Find the node to delete.
    # If the value specified by delete does not exist in the tree, then don't change the tree.
    # If you find the node and ...
    #  a) The node has no children, just set it's parent's pointer to Null.
    #  b) The node has one child, make the nodes parent point to its child.
    #  c) The node has two children, replace it with its successor, and remove
    #       successor from its previous location.
    # Recall: The successor of a node is the left-most node in the node's right subtree.
    # Hint: you may want to write a new method, findSuccessor() to find the successor when there are
      two children

    root = self.getRoot()

    deleted_node = self.deleteTheNode(root, data)


def deleteTheNode(self, root, data):
    # This is a function I created to assist the delete function so I can take in the current root
      (which
    # changes as you can see below) to accurently delete the desired node.

    if root == None:
        return root

    if data < root.getData():
        root.setLeftChild(self.deleteTheNode(root.getLeftChild(), data))

    elif data > root.getData():
        root.setRightChild(self.deleteTheNode(root.getRightChild(), data))
    else:
        if root.getLeftChild() == None: # If there is no left child, then that means there are either
            no children,
        # or it has just one right child.
            temp_node = root.getRightChild()
            root = None
            return temp_node
        elif root.getRightChild() == None: # No children.
            temp_node = root.getLeftChild()
            root = None
            return temp_node

        # This is a node with two children

        temp_node = self.findSuccessor(root.getRightChild())

        root.setData(temp_node.getData())

        # delete the successor node

        root.setRightChild(self.deleteTheNode(root.getRightChild(), temp_node.getData()))

    return root
```

```
def findSuccessor(self, aNode):
    # This is a new method to find the successor when there are two children and it does so
    # by looping through the node when it definitly has two children (aka when there is
    # a left child, you know there is an automatic right child so there are gaurenteed
    # two children) and then you keep going to the left child and checking.

    # Successor = smallest value greater than the node aka the next # after the node.

    current_node = aNode

    while(current_node.getLeftChild() != None):
        current_node = current_node.getLeftChild()
    return current_node
```