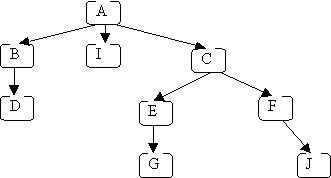
# Pythian Developer Challenge Denver Green : greewden@gmail.com

Please note that in the answers below, I am assuming that a subtree of a tree *T* is a tree consisting of a node in *T* and all of its descendants in *T*.

That is for the tree:  


The tree formed by nodes C, E, F, G, and J would be considered a subtree as it includes all of the descendants of node C, however the tree formed by nodes C, E, and F would not be considered a subtree as it does not include all of the descendants of node C.

1. The code as defined will not work in all scenarios because the pre-order function used to generate the string representation of the tree is not a one-to-one function. A function f is one-to-one if f(a) = f(b) implies that a = b. However for the pre-order function two **different** subtrees can result in the same string representation.  
     
   For example for the subtree  
    [A]  
    [C] [D]  
    [G] [H] [I]  
     
   the pre-order function produces the string “ACGHDI” and for the subtree  
    [A]  
    [C]  
    [G]  
    [H]  
    [D] [ I]  
   the pre-order function also produces the string “ACGHDI”. This behaviour would lead to false positives when determining if a specific subtree is contained in a tree.
2. During the conversion of the tree into a string, information about the structure of the tree is lost, and this is why the pre-order function is not a one-to-one function. For the DOM information being modeled the important structural information that needs to be preserved are the parents of the nodes. Below is a new stringFromPreOrder function that includes parent information in the string generated.  
     
   function stringFromPreOrder(tree: DomNode, parent: string): string {  
    if (!tree) {  
    return "";  
    }  
     
    return parent + ":" + tree.value +   
    stringFromPreOrder(tree.left, tree.value) +   
    stringFromPreOrder(tree.right, tree.value);  
   }
3. Test cases  
   **const *dom***: DomNode = {  
    **value**: **"root"**,  
    **left**: {  
    **value**: **"a"**,  
    **left**: {  
    **value**: **"c"**,  
    **left**: {  
    **value**: **"g"** },  
    **right**: {  
    **value**: **"h"** }  
    },  
    **right**: {  
    **value**: **"d"**,  
    **left**: {  
    **value**: **"i"** }  
    }  
    },  
    **right**: {  
    **value**: **"b"**,  
    **left**: {  
    **value**: **"e"**,  
    **right**: {  
    **value**: **"j"**,  
    **left**: {  
    **value**: **"k"** },  
    **right**: {  
    **value**: **"l"** }  
    }  
    },  
    **right**: {  
    **value**: **"f"** }  
    }  
   }  
     
   *// vdom is a subtree rooted at node a. The string produced for vdom when using a pre-order traversal will be the same  
   // as the string produced for vdom3.***const *vdom***: DomNode = {  
    **value**: **"a"**,  
    **left**: {  
    **value**: **"c"**,  
    **left**: {  
    **value**: **"g"** },  
    **right**: {  
    **value**: **"h"** }  
    },  
    **right**: {  
    **value**: **"d"**,  
    **left**: {  
    **value**: **"i"** }  
    }  
   }  
     
   *// vdom2 is a partial subtree rooted at node A however it does not include all of the descendants of node A.***const *vdom2***: DomNode = {  
    **value**: **"a"**,  
    **left**: {  
    **value**: **"c"** },  
    **right**: {  
    **value**: **"d"** }  
   }  
     
   *// vdom3 is not a subtree of dom however the string produced for vdom3 when using a pre-order traversal  
   // will be the same as the string produced for vdom.***const *vdom3***: DomNode = {  
    **value**: **"a"**,  
    **left**: {  
    **value**: **"c"**,  
    **left**: {  
    **value**: **"g"**,  
    **left**: {  
    **value**: **"h"**,  
    **left**: {  
    **value**: **"d"** },  
    **right**: {  
    **value**: **"i"** }  
    }  
    }  
    }  
   }  
     
   *// vdom4 is not a subtree of dom.***const *vdom4***: DomNode = {  
    **value**: **"c"**,  
    **left**: {  
    **value**: **"g"**,  
    **left**: {  
    **value**: **"h"** }  
    },  
    **right**: {  
    **value**: **"a"**,  
    **left**: {  
    **value**: **"i"** },  
    **right**: {  
    **value**: **"d"** }  
    }  
   }  
     
   **function** *validSubTreeTest*(): **boolean** {  
    *// vdom provided is in dom. This test should return true.* **return** *isSubtree*(***dom***, ***vdom***);  
   }  
     
   **function** *invalidSubTreeTest4*(): **boolean** {  
    *// vdom provided is in dom. This test should return false.* **return** *isSubtree*(***dom***, ***vdom2***);  
   }  
     
   **function** *validSubTreeTest2*(): **boolean** {  
    *// vdom provided is null.* **return** *isSubtree*(***dom***, **null**);  
   }  
     
   **function** *invalidSubTreeTest*(): **boolean** {  
    *// vdom provided is not in dom. This test should return false.* **return** *isSubtree*(***dom***, ***vdom3***);  
   }  
     
   **function** *invalidSubTreeTest2*(): **boolean** {  
    *// vdom provided is not in dom. This test should return false.* **return** *isSubtree*(***dom***, ***vdom4***);  
   }  
     
   **function** *invalidSubTreeTest3*(): **boolean** {  
    *// dom provided is null.* **return** *isSubtree*(**null**, ***vdom***);  
   }  
     
   ***console***.log(**"Valid sub-tree test 1."**);  
   ***console***.log(*validSubTreeTest*());  
   ***console***.log(**"Valid sub-tree test 2."**);  
   ***console***.log(*validSubTreeTest2*());  
   ***console***.log(**"Invalid sub-tree test 1."**);  
   ***console***.log(*invalidSubTreeTest*());  
   ***console***.log(**"Invalid sub-tree test 2."**);  
   ***console***.log(*invalidSubTreeTest2*());  
   ***console***.log(**"Invalid sub-tree test 3."**);  
   ***console***.log(*invalidSubTreeTest3*());  
   ***console***.log(**"Invalid sub-tree test 4."**);  
   ***console***.log(*invalidSubTreeTest4*());
4. To make the tree a k-ary tree instead of a binary tree, replace the left and right attributes of the DomNode interface with an attribute that is an array of DomNodes.  
     
   interface DomNode {  
    value: string;  
    children: Array<DomNode>  
   }
5. An alternative to representing the trees as strings is to actually traverse the tree representing the DOM and for each tree rooted at the current node compare it to the tree representing the virtual DOM. The algorithm which runs in O(n2) time is shown below.  
     
   **function** *isSubtree*(dom: DomNode, vdom: DomNode): **boolean** {  
    *// Virtual DOM is null; return true as a null vdom will always be found in the  
    // DOM.* **if** (vdom === undefined || vdom === **null**){  
    **return true**;  
    }  
     
    *// DOM is null; return false as the virtual DOM cannot be found in a null DOM.* **if** (dom === undefined || dom === **null**) {  
    **return false**;  
    }  
    *// Check to see if the tree rooted at dom contains the tree rooted at vdom.* **if** (*areIdentical*(dom, vdom)) {  
    **return true**;  
    }  
     
    *// The tree rooted ad dom does not contain the tree  
    // rooted at vdom. Check to see if the trees rooted  
    // at the children of dom contain the tree rooted  
    // at vdom.* **return** *isSubtree*(dom.**left**, vdom) || *isSubtree*(dom.**right**, vdom);  
   }  
     
   **function** *areIdentical*(tree: DomNode, subTree: DomNode): **boolean** {  
    *// The order of the two if statements below is important. The AND check must be   
    // done before the OR check.* **if** ((subTree === undefined || subTree === **null**) &&   
    (tree === undefined || tree === **null**)){  
    **return true**;  
    }  
     
    **if** ((subTree === undefined || subTree === **null**) ||   
    (tree === undefined || tree === **null**)) {  
    **return false**;  
    }  
     
    *// Compare the nodes in both trees.* **return** (tree.**value** === subTree.**value** &&   
    *areIdentical*(tree.**left**, subTree.**left**) &&   
    *areIdentical*(tree.**right**, subTree.**right**));  
   }