Part A

```
A <-- Array // An array that has enough space to contain all elements in the tree
count <-- 0

// Traverse the tree in order and put values in an array
function inorder_traversal(tree):
    if (tree is not NULL):
        if (tree.val1 is not NULL):
            inorder_traversal(tree.child0)
            A[count] = tree.val1
            count <-- count + 1
            inorder_traversal(tree.child1)

         if (tree.val2 != NULL):
            inorder_traversal(tree.child2)
            A[count] = tree.val2
            count <-- count + 1

         if (tree.val3 != NULL):
            inorder_traversal(tree.child4)
            A[count] = tree.val3
            count <-- count + 1

function find_median(A[0...n]):
    return A[(n + 1) /2]
```

Part B

```
function count_descendants(node):
    num_descendant <-- 0
    for not null child in node:
        num_decendant <-- num_descendant + child.num_vals + child.num_descendant
    return num_descendant
```

Part C

```
total_val_count <-- tree.num_descendants + tree.num_vals // number of values in the tree
target_count <-- (total_val_count + 1) / 2 // index of the median
temp = 0 // used to store number of values of each child

// check through each child and value in each level
// to find where the median index is located
function find_median_efficient(tree):
    if (tree.child0 != NULL):
        temp = tree.child0.num_descendants + tree.child0.num_vals
    if (total_val_count - temp < target_count):
        return find_median_efficient(tree.child0)
    else if (total_val_count - temp == target_count):
        return tree.val0
    total_val_count = total_val_count - temp - 1
    temp = 0

    if (tree.child1 != NULL):
        temp = tree.child1.num_descendants + tree.child1.num_vals
    else if (total_val_count - temp < target_count):
        return find_median_efficient(tree.child1)
    else if (total_val_count - temp == target_count):
        return tree.val1
    total_val_count = total_val_count - temp - 1
    temp = 0

    if (tree.child2 != NULL):
        temp = tree.child2.num_descendants + tree.child2.num_vals
    else if (total_val_count - temp < target_count):
        return find_median_efficient(tree.child2)
    else if (total_val_count - temp == target_count):
        return tree.val2
    total_val_count = total_val_count - temp - 1
    temp = 0

    // Median must be in the last child
    return find_median_efficient(tree.child3)
```

Part D

Since `num_descendants` and `num_vals` are computed before running the find medain algorithm and included in the tree structure, it will only take $O(1)$ time to get them.

In the best case, in which the median is the first value of the root, it will only take $O(1)$ time to get the result.

Since there are at most 4 children of each node, it will take $O(1)$ time to travel through them

In the worst case, the median is in the leaf node of the longest branch of the tree, and that branch is located in the right most child. Since travelling through children in a level will only take $O(1)$ time, the algorithm will take $\theta(h)$ time to travel to the bottom of the tree, where h represents the height of the tree.

In terms of space complexity, neglecting the memory usage for recursion, the algorithm will take $O(1)$ space since it doesn't require any external storage besides several variables that store temporary data.