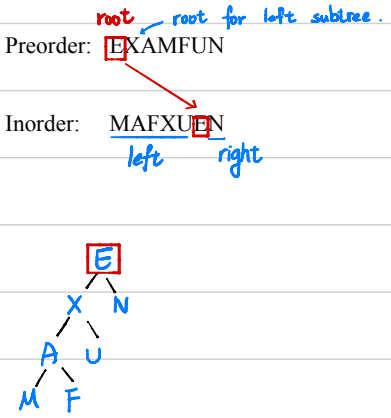
①

```
// Pre-processing
array[26]                    ← serve as an map to imply location of character.

for index in 0 to Inorder.length() - 1:
    char <- Inorder[index]
    array[char] = index
end for

// Recursion
Preorder_Index <- 0

Function Constuct (Inorder, Preorder, Inorder_left, Inorder_right)
    if Inorder_left > Inorder_right:
        return NULL
    end if

    root_Node <- Node(Preorder[Preorder_Index])
    Preorder_Index <- Preorder_Index + 1        ← root

    if Inorder_left == Inorder_right:
        return root_Node
    end if

    Inorder_Index <- array[Preorder[Preorder_Index]]   ← find root in Inorder : (left) root (right)

    root_Node->left <- Construct(Inorder, Preorder, Inorder_left, Inorder_Index - 1)
    root_Node->rightt <- Construct(Inorder, Preorder, Inorder_Index + 1, Inorder_right)

    return root_Node

end function

// After Preprocessing
Tree <- Construct(Inorder, Preorder, 0, Inorder.length() - 1)
```



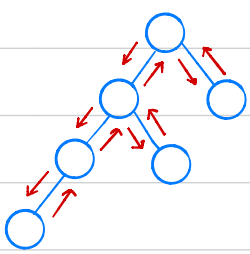
②

```
vector <Node*> parent

Function euler_tour_counter(root)
    if root is NULL:
        return

    if root->left is not NULL:
        parent.push_back(root)
        for node in parent:
            node->descendant <- node->descendant + 1
        parent.pop_back()

    if root->right is not NULL:
        parent.push_back(root)
        for node in parent:
            node->descendant <- node->descendant + 1
        parent.pop_back()

    return

End function
```

↑ Euler Tour

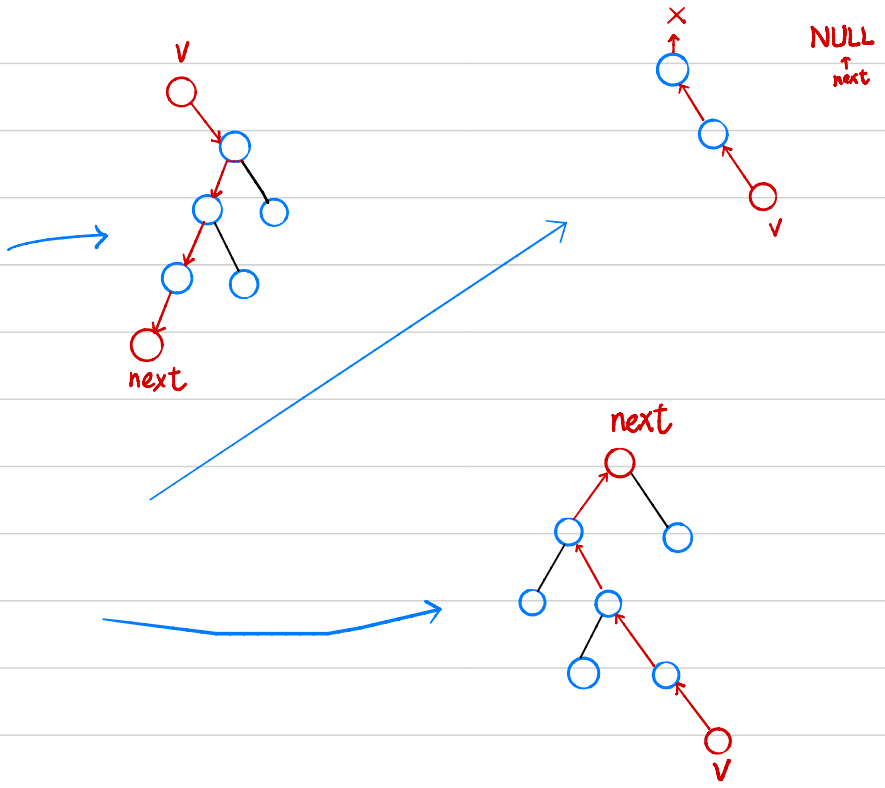
```
Function dfs(root)
    if root is NULL:
        return 0
    root->descendant = dfs(root->left) + dfs(root->right)

    return root->descendant

End function
```

↑ DFS

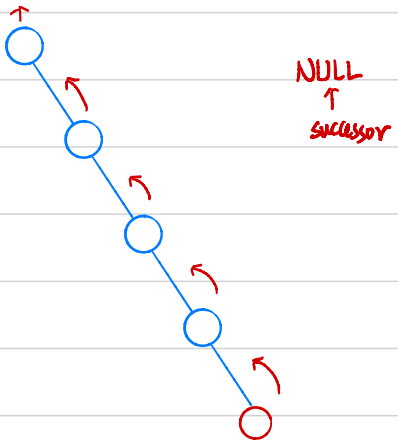


③

```
Function inorderNext(v)
    d <- v

    if d == NULL:
        return NULL
    end if

    if d->right != NULL:
        d <- d->right

        while d->left != NULL:
            d <- d->left
        end while

        return d
    end if

    while d != NULL:
        p <- d->parent

        if p == NULL OR p->left == d:
            return p
        else:
            d = p
        end if
    end while

    return NULL
End function
```



The worst cast would take N iteration (N is the # nodes) O(N). Consider a tree is tilted and only exist right leaf with left child pointer points to NULL, the tree won't be able to find its sucessor of the rightmost nodes and you still needs N traversal to confirm this. So the worst running case is O(N)

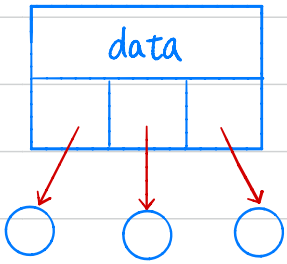


Need N traversal ⇒ O(N)

④

```
Function tab(layer)
    for i = 1 to layer:
        print("\t")
    end for
End function

Function Preorder(root, layer)
    tab(layer)
    print(root->data)

    if root has child node:

        print("(\n")

        for node in root's children list:
            Preorder(node, layer + 1)
        end for

        tab(layer)
        print(")\n")
    end if
End function
```

```
Structure Node{
    data
    list of Node's child
}
```



← something like Preorder

← from leftmost to rightmost child

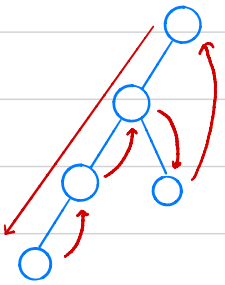
⑤

```
Function inorder(root)
    Stack
    curr <- root

    while(curr is not NULL OR Stack is not empty):

        while(curr is not NULL):
            Stack.push(curr)
            curr <- curr->left
        end while

        curr <- Stack.top()
        Stack.pop()

        print(curr.data)

        curr <- curr->right
    end while
End function
```

← reach leftmost and store previous

← get leftmost

← keep trying until find a node with right leaf.

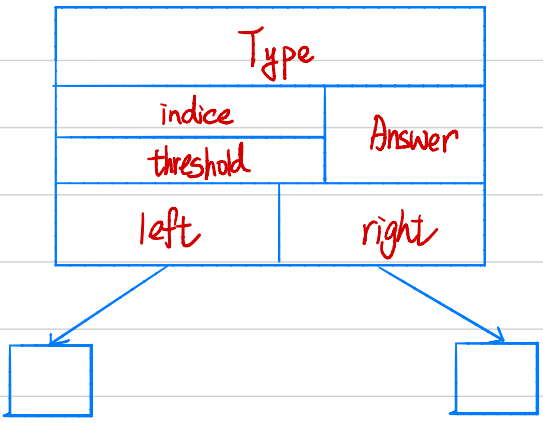


2. Decision Tree

① Before we start finding the best threshold, we need to determine whether these M examples' confusion is greater than epsilon. We need the total number of Yes / No ; let's say it's aYbN. And then I put those examples into a map<float, pair<int, int>> with key the value of V_i (numerical variable) and the value a tuple of (Yes, No) counting the number of example whose numerical variable equals V_i. So I got a sorted sequence (sort by key — numerical variable) and I use additional two variables to record accumulative Yes / No with the iteration of loop. So within the i-th iteration, accumulative Yes = $yes_1 + yes_2 \cdots + yes_i$. (So as No). Thus, I can get the number of both example's whose numerical variable larger and smaller than current considering threshold with one the accumulative and the other being Total - Accumulative. Thus, for finding the best threshold, I only need $M-1$ iteration among the sorted key $\{V_1, V_2 \cdots V_M\}$ So finding the threshold itself is $O(M)$ within sorted numerical variables.

③

```
enum ANSWER{YES, NO};
enum TYPE{CONDITION, RESULT};

class Node{
public:
   Node* left;
   Node* right;

   TYPE type;      ← specify whether the tree is a condition or Y/N.
   int indice;
   double threshold;
   ANSWER ans;     ← Yes/No.

   // Constructor
   Node(Node* L, Node* R, TYPE T):
     left(L), right(R), type(T){}
};
```



* attr [indice] < threshold ⇐ Type : Condition

* return Ans ⇐ Type : Result.

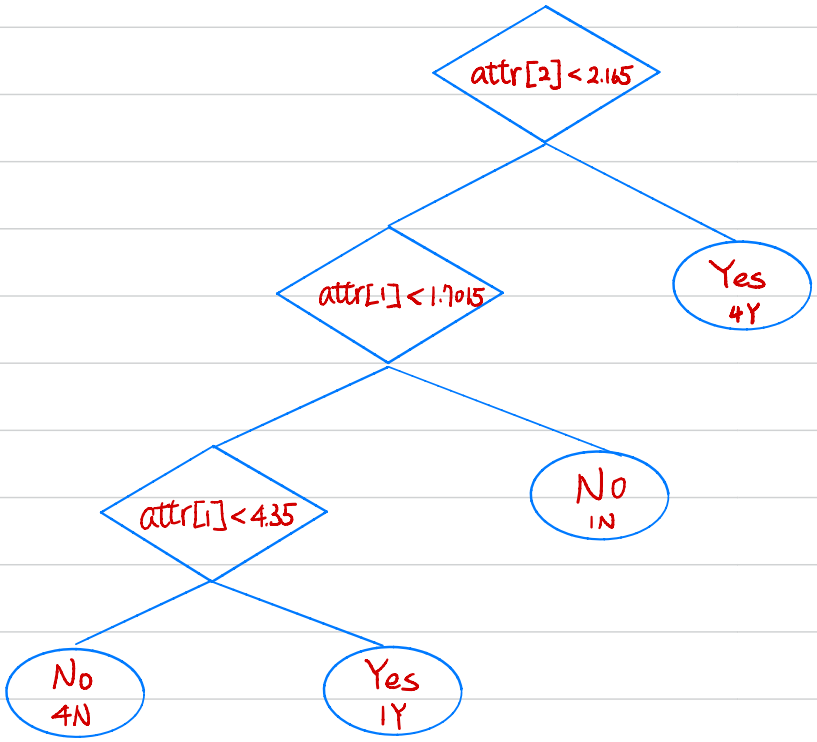
④

Example Data:

+1 1:3.4 2:7.14 3:2.99
+1 1:24.2 3:1.22
+1 1:0.14 2:1.11 3:42.33
+1 1:3
+1 1:32.32 2:0.03 3:2
-1 1:0.003 2:145.22 3:0.03
-1 1:23.3 2:33.4 3:42.3
-1 1:5.3 2:42
-1 1:53 2:3.22 3:0.02
-1 1:11.69 2:3.52 3:42.3

Tree:

```
int tree_predict(double *attr){
  if(attr[2] < 2.165000){
        return 1;
  }else{
        if(attr[1] < 1.701500){
              return -1;
        }else{
              if(attr[1] < 4.350000){
                    return 1;
              }else{
                    return -1;
              }
        }
  }
}
```



For data which enters the condition (ex. attr[2] < 2.165) would be in the right subtree and the other in the left. And to decide the threshold, I simply go through all field (1 to 3) and store the numerical variables in a sorted list. Than take $\frac{1}{2}(V_i + V_{i+1})$ as potential threshold to get the total confusion; the one with the smallest total confusion will be the proper threshold.

So for any testcase, the tree will simply follow the decision tree from the condition ◇ (fulfill → right ; otherwise → left) until hit a result ⬭. And it will be the final decision.

⑤

random : 13.2 < min_confusion : 15.8 < max_confusion : 18.2

① minimum confusion : it tries to find a threshold s.t. same labels are in the same groups as they can. By doing so, Yes will be in one group and No in other as clearly as they can.

② Maximum confusion : it is mostly the same as minimum confusion, aiming to seperate Yes and No as it could.

③ random branching : Since min/Max confusion tend to make two group as clearly as they can, random branching will follow normal distribution with splitting the data into two group with similar size the biggest possibility.

However, the property of data affects a lot !