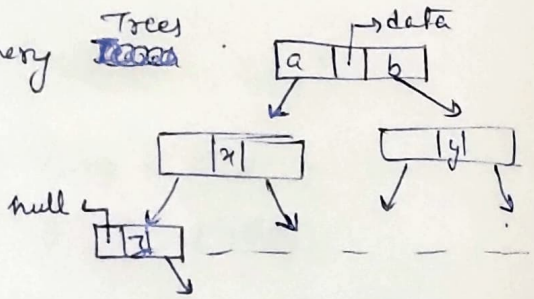
Tree Data Structure (Addition Java Notes).

① → To represent hierarchy.

② → Applications

- In a company hierarchy.
- In folder system.
- XML/HTML content (JSON objects).
- In OOPS (Inheritence).
- Binary Search Tree.
- Binary Heap.
- B & B+ Trees in DBMS.
- Trie.

③ → In Binary Trees



a → left pointer
b → right pointer
in C/C++
But in Java they are called left reference & right reference.

④ Binary Tree Implementation.

```
class Node
{ int key;
  Node left;
  Node right;
  Node(int k) { key = k; }
}
```

↳ class structure

Similar to impl of linked list.

Traversal
- Breadth First Traversal
- Depth First Traversal
  - Inorder
  - Preorder
  - Postorder

→ Traversal given in coolies notes. Pls refer them.

Main code

```
Node tree1 = new Node(x);
tree1.left = Node(x1);
tree2.right = Node(x2);
tree1.left.left = Node(x3)
: etc.
```

Instead of tree1, root used usually

Non recursive.

Traversal

Inorder

```
void inorder(Node root)
{   if(root != null)
    { inorder(root.left); (1)
      System.out.println(root.key) (2)
      inorder(root.right); (3)
    }
}
```

O(n) = Time.
Space = height
= O(height)

preorder → (2) (1) then (3)
postorder → (1) (3) and (2).

Inserting items in tree recursively

In BinaryTree1 class

```
private Node addRecursive(Node current, int value)
{
    if (current == null)
    { return new Node(value); }

    if (value < current.value)
    { current.left = addRecursive(current.left, value); }
    else if (value > current.value)
    { current.right = addRecursive(current.right, value); }
    else { return current; }

    return current
}

public void add(int value)
{
    root = addRecursive(root, value);
}
```

then,

```
BinaryTree1 bt = new BinaryTree1();
bt.add(6);
bt.add(4);
bt.add(8);
```

finding an element.

```
private boolean containsNodeRecursive(Node current, int value)
{
    if (current == null)
    { return false; }

    if (current.value == value)
    { return true; }

    return ( if (value < current.value)
               { containsNodeRecursive(current.left, value);
               }
             else
               { contains------(current.right, value);
               }
           );

}

public boolean containsNode(int value)
{ return containsNodeRecursive(root, value); }

BinaryTree1 bt = new BinaryTree1();
bt.containsNode(6);
```

Deleting an element

- Deleting an element is very similar to searching code. as we can't delete without searching, So let's first write code Part 1.

Part 1.

```
private Node deleteRecursive (Node current, int value)
{
    if (current == null)
    { return null; }

    if (value == current.value)
    { (Part 2) }

    if (value < current.value)
    { current.left = deleteRecursive (current.left, value);
      return current; }

    current.right = deleteRecursive (current.right, value);
    return current;
}
```

Write fn and calling in main, similar to last page i.e in Searching.

(Part 2).

→ if(current.left == null && current.right == null)
{ return null; }

if Node has No child.

→ if(current.right == null)
{ return current.left; }

if(current.left == null)
{ return current.right; }

Has only 1 child.

for 2 children → we use smallest value of right sub tree. - of the node to be deleted.

```
private int findSmallestValue(Node root)
{ return root.left == null ? root.value : findSmallestValue(root.left);
}

int smallestValue = findSmallestValue(current.right);
current.value = smallestValue;
current.right = deleteRecursive(current.right, smallestValue);
return current;
```