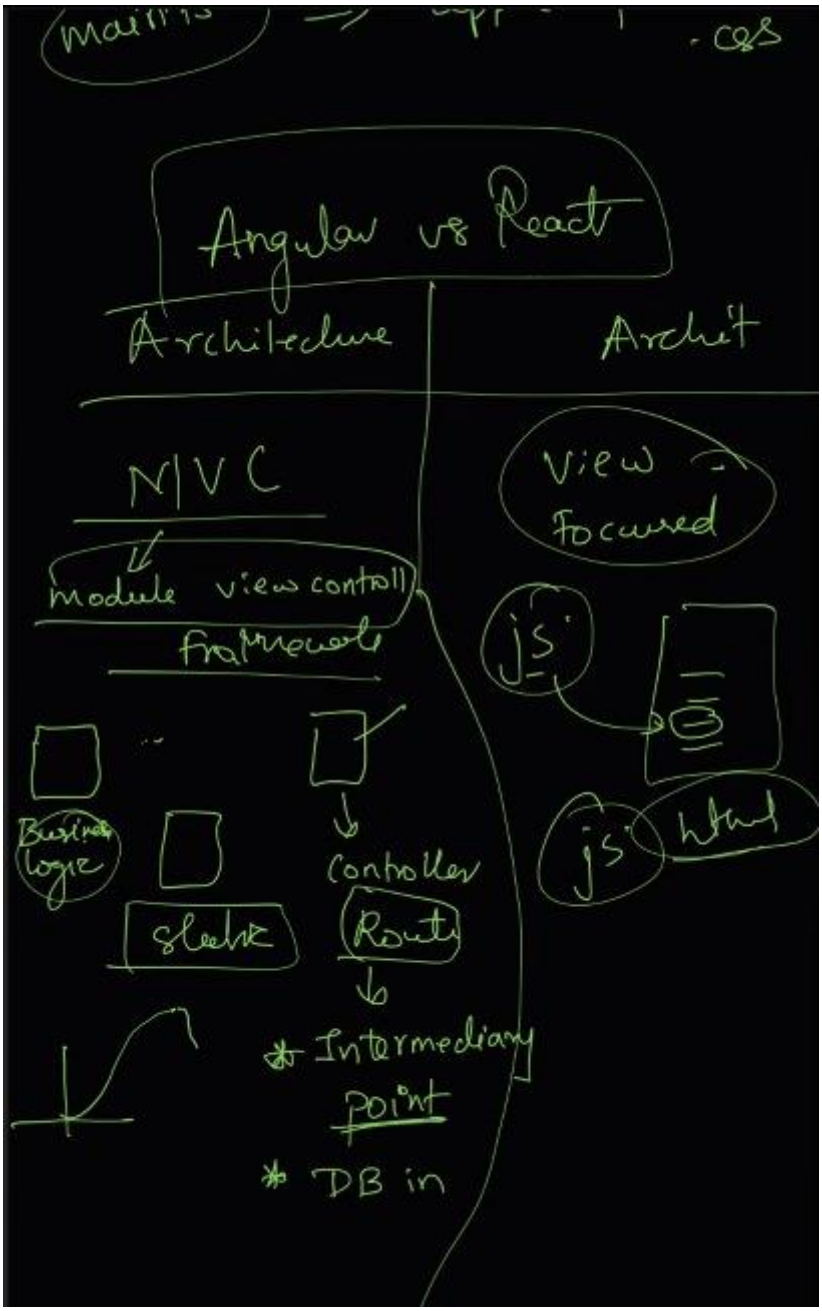
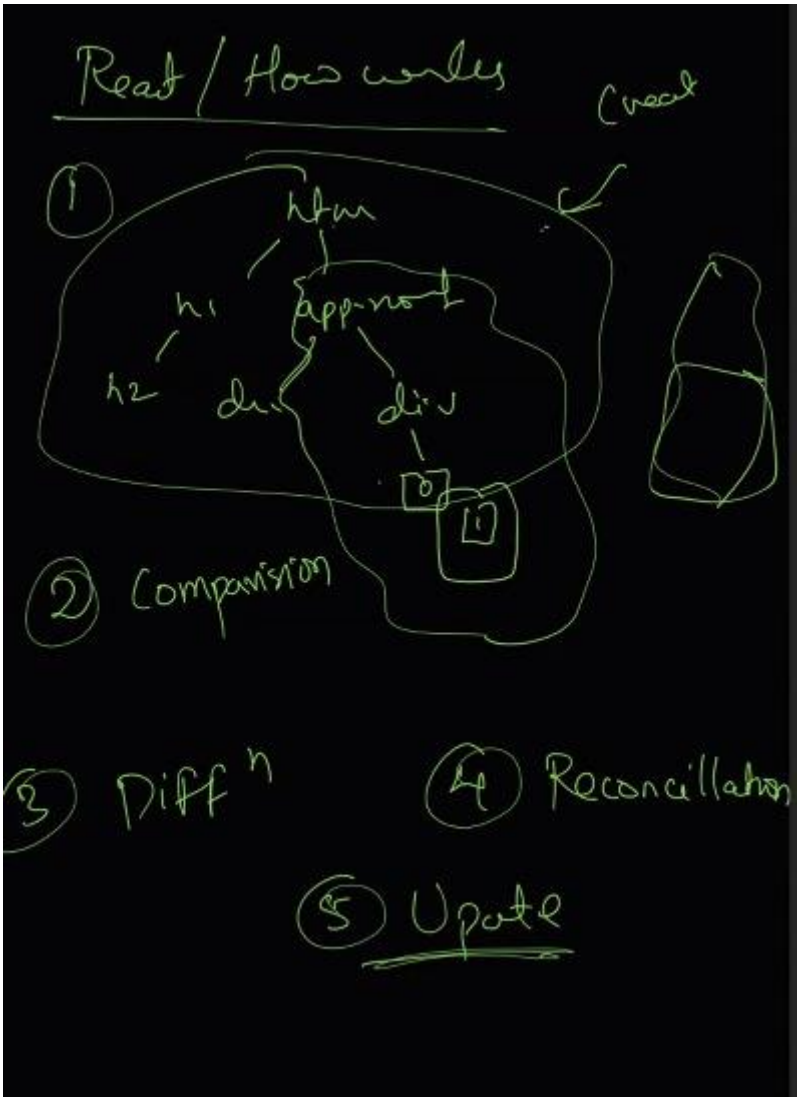
Angular vs React:



- Angular is more MVC focused whereas React is more view focused.
- by default language in angular is type script. whereas in react it has option of supporting javaScript and typeScript both.
- angular uses actual dom. Wheras react uses virtual dom.

virtual dom is a lightweight version of actual DOM (document object model) and it works in 5 steps.



- create virtual dom .
- comparison from old dom where changes have been made .
- then find difference and report it.
- then reconciliation to that it compares with real dom and virtual dom and goes to that specific element header where changes need to be made .
- and finally batch update for that specific div to actually make the changes .

whereas in angular it refreshes or updates the whole container .

if u have frequent updates like 1000's of updates then we use react and but if our page is static and changes are limited then we use angular .

Then we discuss data structure's within python:

Firstly we wrote code for implementing stack:

```python
#Code a stack prompem to cover all concepst of stack and its appication
import math

class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if self.is_empty():
            raise IndexError("Pop from an empty stack")
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            raise IndexError("Peek at an empty stack")
        return self.items[-1]

    def size(self):
        return len(self.items)

    def __str__(self):
        return str(self.items)

    def __repr__(self):
        return repr(self.items)

    def __len__(self):
        return len(self.items)

    def __contains__(self, item):
        return item in self.items

    def __getitem__(self, index):
        return self.items[index]

    def __setitem__(self, index, value):
        self.items[index] = value
```

```
    def __delitem__(self, index):
        del self.items[index]

    def __iter__(self):
        return iter(self.items)

    def __reversed__(self):
        return reversed(self.items)

    def __eq__(self, other):
        return self.items == other.items

    def __ne__(self, other):
        return self.items != other.items

    def __gt__(self, other):
        return self.items > other.items

    def __ge__(self, other):
        return self.items >= other.items

    def __lt__(self, other):
        return self.items < other.items

    def __le__(self, other):
        return self.items <= other.items

    def __hash__(self):
        return hash(tuple(self.items))

    def __add__(self, other):
        return self.items + other.items

    def __iadd__(self, other):
        self.items += other.items
        return self

    def __mul__(self, other):
        return self.items * other

    def __imul__(self, other):
        self.items *= other
        return self

    def __truediv__(self, other):
        if other == 0:
            raise ZeroDivisionError("Division by zero")
        return [x / other for x in self.items]
```

```
    def __itruediv__(self, other):
        if other == 0:
            raise ZeroDivisionError("Division by zero")
        self.items = [x / other for x in self.items]
        return self

    def __floordiv__(self, other):
        if other == 0:
            raise ZeroDivisionError("Division by zero")
        return [x // other for x in self.items]

    def __ifloordiv__(self, other):
        if other == 0:
            raise ZeroDivisionError("Division by zero")
        self.items = [x // other for x in self.items]
        return self

    def __mod__(self, other):
        if other == 0:
            raise ZeroDivisionError("Modulo by zero")
        return [x % other for x in self.items]

    def __imod__(self, other):
        if other == 0:
            raise ZeroDivisionError("Modulo by zero")
        self.items = [x % other for x in self.items]
        return self

    def __pow__(self, other):
        return [x ** other for x in self.items]

    def __ipow__(self, other):
        self.items = [x ** other for x in self.items]
        return self

    def __neg__(self):
        return [-x for x in self.items]

    def __pos__(self):
        return [+x for x in self.items]

    def __abs__(self):
        return [abs(x) for x in self.items]

    def __round__(self, n=None):
        return [round(x, n) for x in self.items]
```

```
    def __ceil__(self):
        return [math.ceil(x) for x in self.items]

    def __floor__(self):
        return [math.floor(x) for x in self.items]

if __name__ == "__main__":
    stack = Stack()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print(stack)
    print(stack.pop())
    print(stack)
    print(stack.peek())
    print(stack.size())
    print(stack.is_empty())
    print(stack.items)
    print(stack.items[0])
    print(stack.items[-1])
    print(stack.items[1:3])
    print(stack.items[:3])
    print(stack.items[1:])
    print(stack.items[::2])
    print(stack.items[::-1])
    print(stack.items[::-2])
    print(stack.items[::-3])
    print(stack.items[::-4])
    print(stack.items[::-5])
    print(stack.items[::-6])
    print(stack.items[::-7])
    print(stack.items[::-8])
    print(stack.items[::-9])
```

And then we wrote a code to interchange two stacks without using the third stack:

```
#interchange the elements of two stacks without using the third stack
from stack import Stack

def insert_at_bottom(stack, item):
    if stack.is_empty():
        stack.push(item)
        return
    temp = stack.pop()
    insert_at_bottom(stack, item)
    stack.push(temp)

def interchange_stacks(stack1, stack2):
    # Get initial sizes
    size1 = stack1.size()
    size2 = stack2.size()
    # First move all elements from stack1 to bottom of stack2
    for _ in range(size1):
        temp = stack1.pop()
        insert_at_bottom(stack2, temp)
    # Now move the original stack2 elements to stack1
    for _ in range(size2):
        temp = stack2.pop()
        insert_at_bottom(stack1, temp)

if __name__ == "__main__":
    # Test the implementation
    stack1 = Stack()
    stack2 = Stack()

    # Initialize stacks
    stack1.push(1)
    stack1.push(2)
    stack1.push(3)
    stack2.push(4)
    stack2.push(5)
    stack2.push(6)

    print("Before interchange:")
    print("Stack1:", stack1)  # Should print [3, 2, 1]
    print("Stack2:", stack2)  # Should print [6, 5, 4]
    interchange_stacks(stack1, stack2)
    print("\nAfter interchange:")
    print("Stack1:", stack1)  # Should print [6, 5, 4]
    print("Stack2:", stack2)  # Should print [3, 2, 1]
```

and then we wrote the code for tree:

```python
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

    def add_child(self, child):
        self.children.append(child)

    def remove_child(self, child):
        self.children.remove(child)

    def __str__(self):
        return str(self.value)

    def __repr__(self):
        return f"TreeNode({self.value})"

    def __len__(self):
        return len(self.children)

    def __contains__(self, value):
        return value in self.children

    def __getitem__(self, index):
        return self.children[index]

    def __setitem__(self, index, value):
        self.children[index] = value

    def __delitem__(self, index):
        del self.children[index]

    def __iter__(self):
        return iter(self.children)

    def __reversed__(self):
        return reversed(self.children)

    def __eq__(self, other):
        return self.value == other.value

    def __ne__(self, other):
        return self.value != other.value

    def __gt__(self, other):
        return self.value > other.value
```

```
    def __ge__(self, other):
        return self.value >= other.value

    def __lt__(self, other):
        return self.value < other.value

    def __le__(self, other):
        return self.value <= other.value

    def __hash__(self):
        return hash(self.value)

    def __bool__(self):
        return bool(self.value)

    def delete(self, value):
        """Deletes a child node with the given value."""
        for child in self.children:
            if child.value == value:
                self.children.remove(child)
                return
            else:
                child.delete(value)

    def inorder_traversal(self):
        """Inorder traversal of the tree."""
        if self.children:
            for child in self.children:
                child.inorder_traversal()
        print(self.value)

    def preorder_traversal(self):
        """Preorder traversal of the tree."""
        print(self.value)
        if self.children:
            for child in self.children:
                child.preorder_traversal()

    def postorder_traversal(self):
        """Postorder traversal of the tree."""
        if self.children:
            for child in self.children:
                child.postorder_traversal()
        print(self.value)

    def __dir__(self):
        return dir(self.value)
```

```python
if __name__ == "__main__":
    tree = TreeNode("A")
    tree.add_child(TreeNode("B"))
    tree.add_child(TreeNode("C"))
    tree.add_child(TreeNode("D"))
    tree.add_child(TreeNode("E"))
    tree.add_child(TreeNode("F"))
    tree.add_child(TreeNode("G"))
    tree.add_child(TreeNode("H"))
    tree.add_child(TreeNode("I"))
    tree.add_child(TreeNode("J"))
    tree.add_child(TreeNode("K"))
    tree.add_child(TreeNode("L"))
    tree.add_child(TreeNode("M"))
    tree.add_child(TreeNode("N"))
    tree.add_child(TreeNode("O"))
    tree.add_child(TreeNode("P"))
    tree.add_child(TreeNode("Q"))
    tree.add_child(TreeNode("R"))
    tree.add_child(TreeNode("S"))
    tree.add_child(TreeNode("T"))
    tree.add_child(TreeNode("U"))
    tree.add_child(TreeNode("V"))
    tree.add_child(TreeNode("W"))
    tree.add_child(TreeNode("X"))
    tree.add_child(TreeNode("Y"))
    tree.add_child(TreeNode("Z"))
    print(tree)
    tree.inorder_traversal()
    tree.preorder_traversal()
    tree.postorder_traversal()
    print(tree.children)
    print(tree.children[0])
    print(tree.children[1])
    print(tree.children[0].children)
    print(tree.children[1].children)
    print(tree.children[0].children)
    print(tree.children[1].children)
    tree.delete("B")
    tree.inorder_traversal()
    tree.preorder_traversal()
    tree.postorder_traversal()
```

and then we wrote the code for various traversal methods:

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

def insert_inorder(root, values, index=0):
    """Inserts values into a binary tree in inorder fashion."""
    if index < len(values):
        node = TreeNode(values[index])
        root = node

        # Insert left child
        root.left = insert_inorder(root.left, values, 2 * index + 1)

        # Insert right child
        root.right = insert_inorder(root.right, values, 2 * index + 2)

    return root

def inorder_traversal(root):
    """Performs inorder traversal of the binary tree."""
    if root:
        inorder_traversal(root.left)
        print(root.value, end=" ")
        inorder_traversal(root.right)

def preorder_traversal(root):
    """performs preorder traversal of the tree"""
    if root:
        print(root.value, end=" ")
        preorder_traversal(root.left)
        preorder_traversal(root.right)

def postorder_traversal(root):
    """performs postorder traversal of the tree"""
    if root:
        postorder_traversal(root.left)
        postorder_traversal(root.right)
        print(root.value, end = " ")

if __name__ == "__main__":
    values = [1, 2, 3, 4, 5, 6, 7]
    root = None
    root = insert_inorder(root, values)
```

```
    print("Inorder Traversal:")
    inorder_traversal(root)
    print()

    print("Preorder Traversal:")
    preorder_traversal(root)
    print()

    print("Postorder Traversal:")
    postorder_traversal(root)
    print()
```

and then we wrote the code for bst:

```
class BSTNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def insert(self, value):

        if value < self.value:
            if self.left is None:
                self.left = BSTNode(value)
            else:
                self.left.insert(value)
        elif value > self.value:
            if self.right is None:
                self.right = BSTNode(value)
            else:
                self.right.insert(value)

    def search(self, value):

        if self.value == value:
            return True
        elif value < self.value and self.left:
            return self.left.search(value)
        elif value > self.value and self.right:
            return self.right.search(value)
        return False

    def find_min(self):

        current = self
        while current.left:
```

```
            current = current.left
        return current.value

    def delete(self, value):

        if value < self.value:
            if self.left:
                self.left = self.left.delete(value)
        elif value > self.value:
            if self.right:
                self.right = self.right.delete(value)
        else:
            if self.left is None:
                return self.right
            if self.right is None:
                return self.left
            min_larger_node = self.right.find_min()
            self.value = min_larger_node
            self.right = self.right.delete(min_larger_node)
        return self

    def inorder_traversal(self):

        if self.left:
            self.left.inorder_traversal()
        print(self.value, end=" ")
        if self.right:
            self.right.inorder_traversal()

    def preorder_traversal(self):

        print(self.value, end=" ")
        if self.left:
            self.left.preorder_traversal()
        if self.right:
            self.right.preorder_traversal()

    def postorder_traversal(self):

        if self.left:
            self.left.postorder_traversal()
        if self.right:
            self.right.postorder_traversal()
        print(self.value, end=" ")

# Example Usage
if __name__ == "__main__":
    root = BSTNode(50)
```

```
    root.insert(30)
    root.insert(70)
    root.insert(20)
    root.insert(40)
    root.insert(60)
    root.insert(80)

    print("Inorder Traversal:")
    root.inorder_traversal()  # Sorted order

    print("\nPreorder Traversal:")
    root.preorder_traversal()

    print("\nPostorder Traversal:")
    root.postorder_traversal()

    print("Search 100:", root.search(100))

    print("\nDeleting 40...")
    root.delete(40)
    root.inorder_traversal()
```

and then we wrote the code for trie block for dealing with databases:

```
import os
import pickle
import time

class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self, db_file="trie.pkl"):
        self.root = TrieNode()
        self.db_file = db_file
        if os.path.exists(db_file):
            self.load()

    def insert(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        node.is_end_of_word = True
```

```
    def search(self, word):
        node = self.root
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        return node.is_end_of_word
    def starts_with(self, prefix):
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True
    def get_all_words(self, node=None, prefix=""):
        if node is None:
            node = self.root
        words = []
        if node.is_end_of_word:
            words.append(prefix)
        for char, child in node.children.items():
            words.extend(self.get_all_words(child, prefix + char))
        return words
    def save(self):
        with open(self.db_file, "wb") as f:
            pickle.dump(self.root, f)

    def load(self):
        with open(self.db_file, "rb") as f:
            self.root = pickle.load(f)
def build_trie_from_dataset(trie, dataset_file):
    with open(dataset_file, "r") as f:
        for line in f:
            word = line.strip()  # Directly read the word
            trie.insert(word)
    trie.save()
def measure_time_complexity(trie):
    start_time = time.time()
    all_words = trie.get_all_words()
    end_time = time.time()
    elapsed_time = end_time - start_time
    print(f"Found {len(all_words)} words in {elapsed_time:.6f} seconds.")
    return elapsed_time
# Example usage:
trie = Trie("my_trie.pkl")
build_trie_from_dataset(trie, "words.txt") # Your word list file.
measure_time_complexity(trie)
trie.
```