1. create a tree convert into bst

```c
#include <stdio.h>

#include <stdlib.h>


// Define the structure for a node

struct Node {

   int data;          // Data of the node

   struct Node* left;  // Pointer to the left child node

   struct Node* right; // Pointer to the right child node

};


// Function to create a new node

struct Node* createNode(int value) {

   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node)); // Allocate memory for the
new node

   newNode->data = value;   // Set the node's data

   newNode->left = NULL;    // Initialize left child as NULL

   newNode->right = NULL;   // Initialize right child as NULL

   return newNode;

}


// Function to do in-order traversal and store the nodes in an array

void inorderTraversal(struct Node* root, int* arr, int* index) {

   if (root == NULL) {

      return;

   }


   inorderTraversal(root->left, arr, index);  // Traverse left subtree

   arr[(*index)++] = root->data;             // Store node's data

   inorderTraversal(root->right, arr, index); // Traverse right subtree

}
```

```
// Function to sort the array (in ascending order)
void sortArray(int* arr, int n) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = i + 1; j < n; j++) {
            if (arr[i] > arr[j]) {
                // Swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}

// Function to convert the binary tree into a Binary Search Tree (BST)
void arrayToBST(int* arr, struct Node* root, int* index) {
    if (root == NULL) {
        return;
    }

    // Recur to left subtree
    arrayToBST(arr, root->left, index);

    // Assign the next value from the sorted array to the current node
    root->data = arr[(*index)++];

    // Recur to right subtree
    arrayToBST(arr, root->right, index);
}
```

```
// Main function
int main() {
    // Create a binary tree
    struct Node* root = createNode(10);
    root->left = createNode(5);
    root->right = createNode(15);
    root->left->left = createNode(3);
    root->left->right = createNode(7);
    root->right->left = createNode(12);
    root->right->right = createNode(18);

    // Step 1: Traverse the binary tree and store the nodes in an array
    int arr[100];    // Array to store the values
    int index = 0;
    inorderTraversal(root, arr, &index);

    // Step 2: Sort the array
    int n = index;   // Number of elements in the array
    sortArray(arr, n);

    // Step 3: Rebuild the tree to satisfy BST property using the sorted array
    index = 0;  // Reset index to reuse in arrayToBST
    arrayToBST(arr, root, &index);

    // After conversion, the tree is now a Binary Search Tree (BST)
    printf("In-order Traversal of the converted BST: ");
    inorderTraversal(root, arr, &index); // In-order traversal of the BST
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
```

```
    return 0;
}
```

Pre-order

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a node
struct Node {
    int data;          // Data of the node
    struct Node* left;  // Pointer to the left child node
    struct Node* right; // Pointer to the right child node
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node)); // Allocate memory for the new node
    newNode->data = value;   // Set the node's data
    newNode->left = NULL;     // Initialize left child as NULL
    newNode->right = NULL;    // Initialize right child as NULL
    return newNode;
}

// Pre-order traversal (Root, Left, Right)
void preorderTraversal(struct Node* root) {
    if (root == NULL) {
        return;
    }

    // Visit the root node
    printf("%d ", root->data);
```

```
    // Traverse the left subtree
    preorderTraversal(root->left);

    // Traverse the right subtree
    preorderTraversal(root->right);
}

int main() {
    // Create the root node
    struct Node* root = createNode(10);

    // Add children to the root node
    root->left = createNode(5);
    root->right = createNode(15);

    // Add children to the left node
    root->left->left = createNode(3);
    root->left->right = createNode(7);

    // Add children to the right node
    root->right->left = createNode(12);
    root->right->right = createNode(18);

    // Perform pre-order traversal and print the nodes
    printf("Pre-order Traversal: ");
    preorderTraversal(root);
    printf("\n");

    return 0;
}
```

Post-order

```
#include <stdio.h>
```

```c
#include <stdlib.h>

// Define the structure for a node
struct Node {
    int data;           // Data of the node
    struct Node* left;  // Pointer to the left child node
    struct Node* right; // Pointer to the right child node
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node)); // Allocate memory for the new node
    newNode->data = value;   // Set the node's data
    newNode->left = NULL;     // Initialize left child as NULL
    newNode->right = NULL;    // Initialize right child as NULL
    return newNode;
}

// Post-order traversal (Left, Right, Root)
void postorderTraversal(struct Node* root) {
    if (root == NULL) {
        return;
    }

    // Traverse the left subtree
    postorderTraversal(root->left);

    // Traverse the right subtree
    postorderTraversal(root->right);

    // Visit the root node
    printf("%d ", root->data);
}
```

```
int main() {
    // Create the root node
    struct Node* root = createNode(10);

    // Add children to the root node
    root->left = createNode(5);
    root->right = createNode(15);

    // Add children to the left node
    root->left->left = createNode(3);
    root->left->right = createNode(7);

    // Add children to the right node
    root->right->left = createNode(12);
    root->right->right = createNode(18);

    // Perform post-order traversal and print the nodes
    printf("Post-order Traversal: ");
    postorderTraversal(root);
    printf("\n");

    return 0;
}
```

In-order

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a node
struct Node {
    int data;          // Data of the node
    struct Node* left;  // Pointer to the left child node
```

```c
    struct Node* right; // Pointer to the right child node
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node)); // Allocate memory for the new node
    newNode->data = value;   // Set the node's data
    newNode->left = NULL;     // Initialize left child as NULL
    newNode->right = NULL;    // Initialize right child as NULL
    return newNode;
}

// In-order traversal (Left, Root, Right)
void inorderTraversal(struct Node* root) {
    if (root == NULL) {
        return;
    }

    // Traverse the left subtree
    inorderTraversal(root->left);

    // Visit the root node
    printf("%d ", root->data);

    // Traverse the right subtree
    inorderTraversal(root->right);
}

int main() {
    // Create the root node
    struct Node* root = createNode(10);

    // Add children to the root node
```

```
    root->left = createNode(5);
    root->right = createNode(15);

    // Add children to the left node
    root->left->left = createNode(3);
    root->left->right = createNode(7);

    // Add children to the right node
    root->right->left = createNode(12);
    root->right->right = createNode(18);

    // Perform in-order traversal and print the nodes
    printf("In-order Traversal: ");
    inorderTraversal(root);
    printf("\n");

    return 0;
}
```

tree convert into AVL tree

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a node
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
    int height;  // Height of the node for AVL balancing
};

// Function to create a new node
struct Node* createNode(int value) {
```

```
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->left = NULL;
    newNode->right = NULL;
    newNode->height = 1;  // Initially, the height is 1 for a new node
    return newNode;
}

// Function to get the height of a node
int height(struct Node* node) {
    if (node == NULL)
        return 0;
    return node->height;
}

// Function to get the balance factor of a node
int getBalance(struct Node* node) {
    if (node == NULL)
        return 0;
    return height(node->left) - height(node->right);
}

// Function to perform a right rotation (used for balancing)
struct Node* rightRotate(struct Node* y) {
    struct Node* x = y->left;
    struct Node* T2 = x->right;

    // Perform rotation
    x->right = y;
    y->left = T2;

    // Update heights
    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));
```

```
    x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));

    // Return the new root
    return x;
}

// Function to perform a left rotation (used for balancing)
struct Node* leftRotate(struct Node* x) {
    struct Node* y = x->right;
    struct Node* T2 = y->left;

    // Perform rotation
    y->left = x;
    x->right = T2;

    // Update heights
    x->height = 1 + (height(x->left) > height(x->right) ? height(x->left) : height(x->right));
    y->height = 1 + (height(y->left) > height(y->right) ? height(y->left) : height(y->right));

    // Return the new root
    return y;
}

// Function to insert a node in an AVL tree and balance it
struct Node* insertAVL(struct Node* node, int data) {
    if (node == NULL)
        return createNode(data);

    if (data < node->data)
        node->left = insertAVL(node->left, data);
    else if (data > node->data)
        node->right = insertAVL(node->right, data);
    else
```

```
        return node;  // Duplicate values are not allowed in BST

    // Update the height of this ancestor node
    node->height = 1 + (height(node->left) > height(node->right) ? height(node->left) : height(node->right));

    // Get the balance factor to check if this node became unbalanced
    int balance = getBalance(node);

    // Left Left Case
    if (balance > 1 && data < node->left->data)
        return rightRotate(node);

    // Right Right Case
    if (balance < -1 && data > node->right->data)
        return leftRotate(node);

    // Left Right Case
    if (balance > 1 && data > node->left->data) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }

    // Right Left Case
    if (balance < -1 && data < node->right->data) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }

    return node;
}

// In-order traversal to collect node values
void inorderTraversal(struct Node* root, int* arr, int* index) {
```

```c
    if (root == NULL)
        return;
    inorderTraversal(root->left, arr, index);
    arr[(*index)++] = root->data;
    inorderTraversal(root->right, arr, index);
}

// Function to convert an unsorted binary tree into an AVL tree
struct Node* convertToAVL(struct Node* root) {
    int arr[100];  // Assuming the tree has at most 100 nodes
    int index = 0;

    // Step 1: Traverse the binary tree in-order to get all the node values
    inorderTraversal(root, arr, &index);

    // Step 2: Insert the values into an AVL tree (sorted)
    struct Node* newRoot = NULL;
    for (int i = 0; i < index; i++) {
        newRoot = insertAVL(newRoot, arr[i]);
    }

    return newRoot;
}

// Function to perform in-order traversal and print the tree
void printInOrder(struct Node* root) {
    if (root == NULL)
        return;
    printInOrder(root->left);
    printf("%d ", root->data);
    printInOrder(root->right);
}
```

```
int main() {
    // Create a binary tree (not necessarily balanced)
    struct Node* root = createNode(10);
    root->left = createNode(5);
    root->right = createNode(15);
    root->left->left = createNode(3);
    root->left->right = createNode(7);
    root->right->left = createNode(12);
    root->right->right = createNode(18);

    // Print original tree (in-order traversal)
    printf("Original tree (In-order): ");
    printInOrder(root);
    printf("\n");

    // Convert the binary tree into an AVL tree
    struct Node* avlRoot = convertToAVL(root);

    // Print the AVL tree (in-order traversal)
    printf("Converted AVL tree (In-order): ");
    printInOrder(avlRoot);
    printf("\n");

    return 0;
}
```