1. If an initially empty BST is to be inserted with an array sorted by descending order, Then each node in the BST will only have a left child and will look like a slash symbol. Otherwise, each node will only have a right child and will look like a backslash symbol when the array is sorted by ascending order.

2. The running time of operation Member for the BST produced in #1 will be O(n). Since the running time O(log n) of operation Member doesn't apply to a BST where all nodes have only the left child or all nodes have only the right child.

3.function delete and insert: Non-recursive and Recursive:

```
void insert(BST *tree, int X) { /* Non-Recursive */
    while (*tree != NULL) {
        tree = (X < (*tree)->elem)? &(*tree)->LC : &(*tree)->RC;
    }

    if (*tree == NULL) {
        *tree = (BST) malloc(sizeof(ctype));
        (*tree)->LC = (*tree)->RC = NULL;
        (*tree)->elem = X;
    }
}

void insert(BST *tree, int X) {/*Recursive */
    if (*tree == NULL) {
        *tree = (BST) malloc(sizeof(ctype));
        (*tree)->LC = (*tree)->RC = NULL;
        (*tree)->elem = X;
    } else {
        (X < (*tree)->elem)? insert(&(*tree)->LC, X): insert(&(*tree)->RC, X);
    }
}

void delete(BST *tree, int X) {/* Non-Recursive */
    while (*tree != NULL && X != (*tree)->elem) {
        tree = (X < (*tree)->elem)? &(*tree)->LC: &(*tree)->RC;
    }

    if (*tree != NULL) {
        BST delNode = *tree;
        if ((*tree)->LC == NULL && (*tree)->RC == NULL) {
            *tree = NULL;
        } else if ((*tree)->LC == NULL) {
            *tree = (*tree)->RC;
        } else if ((*tree)->RC == NULL) {
```

```
            *tree = (*tree)->LC;
        } else {
            BST *trav = &(*tree)->RC;

            while ((*trav)->LC != NULL)
                trav = &(*trav)->LC;
            delNode = *trav;
            (*tree)->elem = delNode->elem;
            *trav = delNode->RC;
        }
        free(delNode);
    }
}

void delete(BST *tree, int X) {/* Recursive */
    if (*tree != NULL && (*tree)->elem != X) {
        (X < (*tree)->elem)? delete(&(*tree)->LC, X): delete(&(*tree)->RC, X);
    } else {
        BST delNode = *tree;
        if ((*tree)->LC == NULL && (*tree)->RC == NULL) {
            *tree = NULL;
        } else if ((*tree)->LC == NULL) {
            *tree = (*tree)->RC;
        } else if ((*tree)->RC == NULL) {
            *tree = (*tree)->LC;
        } else {
            BST *trav = &(*tree)->RC;

            while ((*trav)->LC != NULL)
                *trav = (*trav)->LC;
            delNode = *trav;
            *trav = (*trav)->RC;
            (*tree)->elem = delNode->elem;
        }
        free (delNode);
    }
}
```

4.
an AVL tree is a self-balancing binary search tree. It was the first such data structure to be invented. In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.

5. Binary Search Tree (BST) is a tree data structure that has nodes that also implements BST where lesser values are to the left and greater values are to the left. On the other hand, Binary Search refers to the algorithm that searches on an ordered or sorted collection. I think that Binary Search is present in BST because of how the BST is made and its traversing algorithm is similar to Binary Search.

6.
```
void preorder(BST tree) {
   if (tree != NULL) {
        printf("%d ", tree->elem);
        preorder(tree->LC);
        preorder(tree->RC);
   }
}

void inorder(BST tree) {
   if (tree != NULL) {
        preorder(tree->LC);
        printf("%d ", tree->elem);
        preorder(tree->RC);
   }
}

void postorder(BST tree) {
   if (tree != NULL) {
        postorder(tree->LC);
        postorder(tree->RC);
        printf("%d ", tree->elem);
   }
}
```