```
Algorithm Function BST_from_PostOrder(POSTWALK, BEG, END)
Given the postorder walk of a binary search tree, this function creates a BST
out of it. The postorder walk is denoted by a list [implemnted as arrays],
POSTWALK. BEG and END are the lower and the upper indices of the walk array.
The functions returns a tree pointer. NEWW denotes a local tree pointer. POS
is a local variables. Array indices starts at 0.
1.  Check validity of array bounds
        If BEG > END
            Return NULL
2.  Create and initialize a tree node
        NEWW <== TNODE
        LCHILD(NEWW) = RCHILD(NEWW) = NULL
        DATA(NEWW) = POSTWALK[END]
3.  Is this the only node in current context
        If BEG = END
            Return NEWW;
4.  Initialize POS
        POS = END - 1
5.
        Repeat While POSTWALK[POS] > POSTWALK[END]
```

```
            POS = POS - 1
            If POS = BEG
                Break
6.  Set up recursion
        LCHILD(NEWW) = Call BST_from_PostOrder(POSTWALK, BEG, POS)
        RCHILD(NEWW) = Call BST_from_PostOrder(POSTWALK, POS+1, END-1)
7. Return the tree
        Return NEWW
```

Algorithm Function **BST_from_PreOrder**(PREWALK, PRL, BEG, END)
Given the preorder walk of a binary search tree, this function creates a BST out of it. The preorder walk is denoted by a list [implemnted as arrays], PREWALK. BEG and END are the lower and the upper indices of the walk array. The functions returns a tree pointer. NEWW denotes a local tree pointer. PRL is the input-output parameter indicating current index in PREWALK. LEN is a local variables that holds length(PREWALK). J is a counter variable. Array indices starts at 0.

```
1.  Check validity of array bounds (Base Case)
        If PRL > LEN Or BEG > END
            Return NULL
```

```
2.  Create and initialize a tree node and increment PRL
        NEWW <== tNODE
        LCHILD(NEWW) = RCHILD(NEWW) = NULL
        DATA(NEWW) = PREWALK[PRL]
        PRL = PRL + 1
3.  Is this the only node in current context
        If BEG = END
            Return NEWW;
4.  Locate the first key > root
        J = BEG
        Repeat While J <= END Step 1
            If PREWALK[J] > DATA(NEWW)
                Break
5.  Set up recursion
        LCHILD(NEWW) = Call BST_from_PreOrder(PREWALK, PRL, (PRL), J-1)
        RCHILD(NEWW) = Call BST_from_PreOrder(PREWALK, PRL, J, END)
6. Return the tree
        Return NEWW
```

```
tree BST_from_PostOrder(int postWalk[], int beg, int end){
    tree neww;
    if(beg > end)
        return NULL;
    neww = (tree) calloc(1, sizeof(tNode));
    neww->data = postWalk[end];
    neww->lchild = neww->rchild = NULL;

    if(beg == end)
        return neww;
    int pos = end - 1;
    while(postWalk[pos] > postWalk[end]){
        pos = pos - 1;
        if(pos == beg)
            break;
    }
    neww->lchild = BST_from_PostOrder(postWalk,beg,pos);
    neww->rchild = BST_from_PostOrder(postWalk,pos+1,end-1);
    return neww;
}
```

```
tree BST_from_PreOrder(int preWalk[], int *prl,
                                    int beg, int end, int len){
    tree neww;
    int j;
    if (*prl > len || beg > end)
        return NULL;
    neww = (tree) calloc(1, sizeof(tNode));
    neww->lchild = neww->rchild = NULL;
    neww->data = preWalk[*prl];
    *prl = *prl + 1;
    if(beg == end)
        return neww;
    /*Locate the first key > root */
    for(j = beg; j <= end; ++j )
        if(preWalk[j] > neww->data )
            break;
    neww->lchild = BST_from_PreOrder(preWalk, prl, *prl, j-1, len);
    neww->rchild = BST_from_PreOrder(preWalk, prl, j, end, len);
    return neww;
}
```

```
Algorithm Function BinTreeFromIn_Pre_Order(INWALK, PREWALK, BEG, END)
Given the inorder walk and the preorder walk of a binary tree, this function
creates a Binary tree from them. The inorder and preorder walks are denoted
by lists [implemnted as arrays], INWALK and PREWALK respectively. BEG and END
are the lower and the upper indices of the walk arrays. The functions returns
a tree pointer. NEWW denotes a local tree pointer. INL and PRL are local
variables representing walk indices. J is the counter variable. Array indices
starts at 0.
1.  Initialize PRL and INL. PRL is a static variable
        PRL = 0
        INL = -1    /**denoting non existing position */
2.  Check validity of array bounds
        If BEG > END
            Return NULL
3.  Create a tree node
        NEWW <== TNODE
4.  Initialize the node to data pointed by PRL and update PRL
        LCHILD(NEWW) = RCHILD(NEWW) = NULL
        DATA(NEWW) = PREWALK[PRL]
        PRL = PRL + 1
```

```
5.  Find the ndex of DATA(NEWW) in INWALK
        J = BEG;
        Repeat While J <= END
            If INWALK[J] = DATA[NEWW]
                INL = J
            J = J + 1
6.  Set up the recursion on Left and Right Subtree
        LCHILD(NEWW) =
            Call BinTreeFromIn_Pre_Order(INWALK, PREWALK, BEG, INL-1)
        RLCHILD(NEWW) =
            Call BinTreeFromIn_Pre_Order(INWALK, PREWALK, INL+1, END)
7.  Return the tree
        Return NEWW
```

Algorithm Function **BinTreeFromIn_Post_Order**
(INWALK, POSTWALK, IBEG, IEND, BEGP, ENDP)

Given the inorder walk and the postorder walk of a binary tree, this function creates a Binary tree from them. The inorder and postorder walks are denoted by lists [implemnted as arrays], INWALK and POSTWALK respectively. IBEG and IEND are the lower and the upper indices of the inWalk array whereas BEGP and ENDP denotes lower and upper indices of PostWalk array. The functions returns a tree pointer. NEWW denotes a local tree pointer. INL is a local variable representing walk indix. J is a counter variable. Array indices starts at 0.

1. Initialize INL.

```
INL = -1    /**denoting non existing position */
```

2. Check validity of array bounds

```
If IBEG > IEND Or ENDP < BEGP
    Return NULL
```

3. Create a tree node

```
NEWW <== TNODE
```

4. Initialize the node

```
LCHILD(NEWW) = RCHILD(NEWW) = NULL
DATA(NEWW) = POSTWALK[ENDP]
```

```
5.  If its the only node of current subtree
        If IBEG = IEND
            Return NEWW
6.  Find the index of DATA(NEWW) in INWALK
        J = IBEG;
        Repeat While J <= IEND
            If INWALK[J] = DATA[NEWW]
                INL = J
            J = J + 1
7.  Set up the recursion on Left and Right Subtree
        LCHILD(NEWW) = Call BinTreeFromIn_Post_Order(INWALK, POSTWALK,
                                    IBEG, INL-1, BEGP, BEGP+INL-IBEG-1)
        RCHILD(NEWW) = Call BinTreeFromIn_Post_Order(INWALK, POSTWALK,
                                    INL+1, IEND, BEGP+INL+IBEG, ENDP-1)
8.  Return the tree
        Return NEWW
```

```
tree createTreeInPre(int inWalk[], int preWalk[], int beg, int end){
    static int prl = 0; int inl = -1, j;
    tree neww;

    if(beg > end)
        return NULL;
    neww = (tree) calloc(1, sizeof(tNode));

    neww->data = preWalk[prl];
    neww->rchild = neww->lchild = NULL;
    prl += 1;

    for(j = beg; j <= end; j++)
        if(inWalk[j] == neww->data)
            inl = j;
    neww->lchild = createTreeInPre2(inWalk, preWalk, beg, inl-1);
    neww->rchild = createTreeInPre2(inWalk, preWalk, inl+1, end);
    return neww;
}
```

```
tree createTreeInPost(int inWalk[], int postWalk[],
                      int ibeg, int iend, int begp, int endp){
    int inl = -1, j;
    tree neww;
    if(ibeg > iend || endp < begp)
        return NULL;
    neww = (tree) calloc(1, sizeof(tNode));
    neww->data = postWalk[endp];
    neww->rchild = neww->lchild = NULL;
    if(ibeg == iend)
        return neww;
    for(j = ibeg; j <= iend; j++)
        if(inWalk[j] == neww->data)
            inl = j;
    neww->lchild = createTreeInPost(inWalk, postWalk, ibeg, inl-1,
                                                    begp, begp+ndx-ibeg-1);
    neww->rchild = createTreeInPost(inWalk, postWalk, ndx+1, iend,
                                                    begp+inl-ibeg, endp-1);
    return neww;
}
```