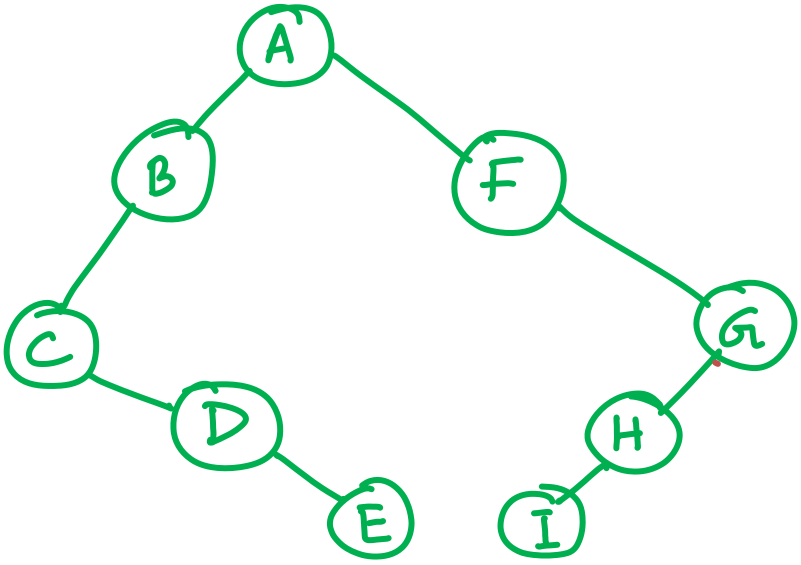
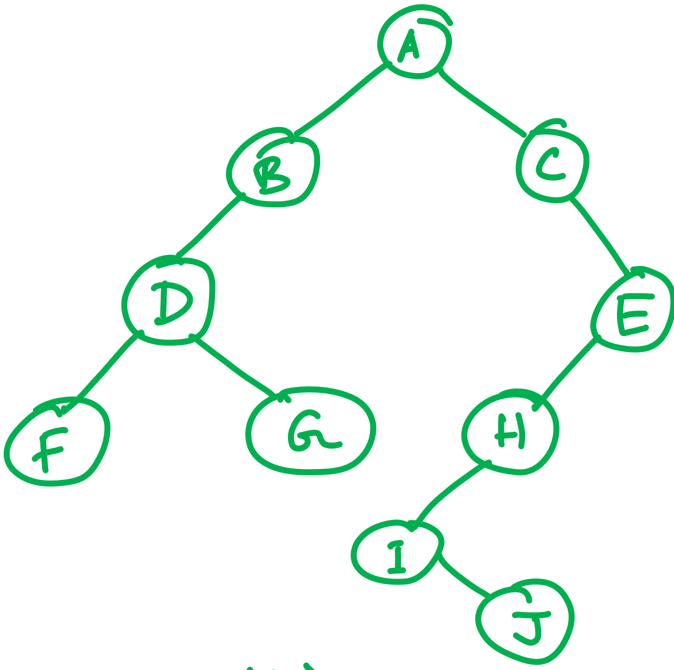
Binary Tree Traversals (Continued)

Dated on 18/12/2021

Find the Preorder, Inorder & Postorder traversal sequences of the following binary Trees.



(a)

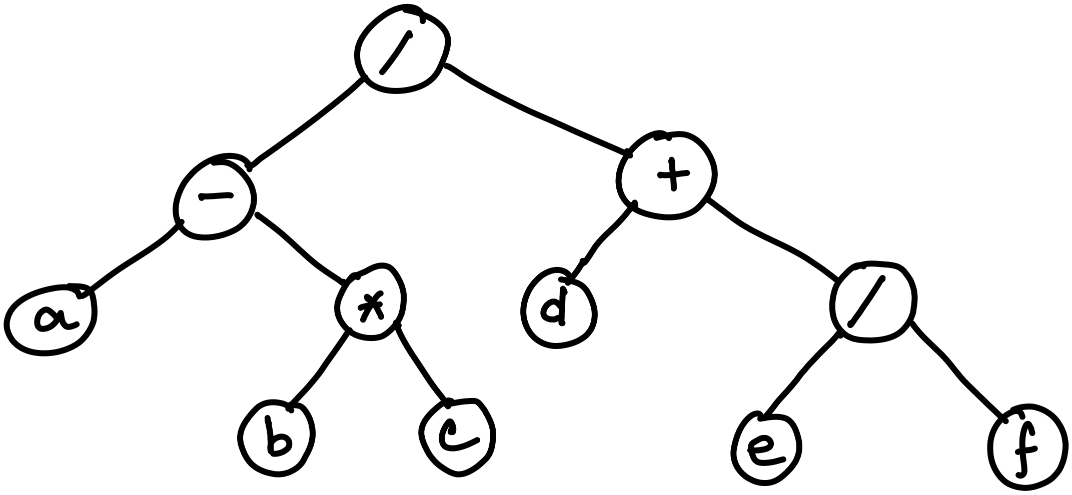


(b)

Preorder: A, B, C, D, E, F, G, H, I
Inorder: C, D, E, B, A, F, I, H, G
Postorder: E, D, C, B, I, H, G, F, A
} (a)

Preorder : A, B, D, F, G, C, E, H, I, J
Inorder : F, D, G, B, A, C, I, J, H, E
Postorder : F, G, D, B, J, I, H, E, C, A
} (b)

Alzebric Expression : $(a-b*c)/(d+e/f)$



Binary tree representing the above given expression.

Preorder Traversal seq.: $/-a*bc+d/ef$ ⇒ Prefix form

Inorder Traversal seq.: $a-b*c/d+e/f$ ⇒ Infix form

Postorder Traversal seq.: $abc*-def/+/$ ⇒ Postfix form

Therefore, for any given mathematic expression, we can convert it into prefix or postfix forms using the binary tree representation of that expression and finding out the preorder and postorder traversal sequences respectively.

Homework:

Find out the prefix and postfix forms of the following expressions using the binary tree representation and its traversal sequences.

(a) $(a*b-c*d)/e+f$

(b) $a*b(c+d/f)$

Binary Search Tree

Dated 21|12|2021

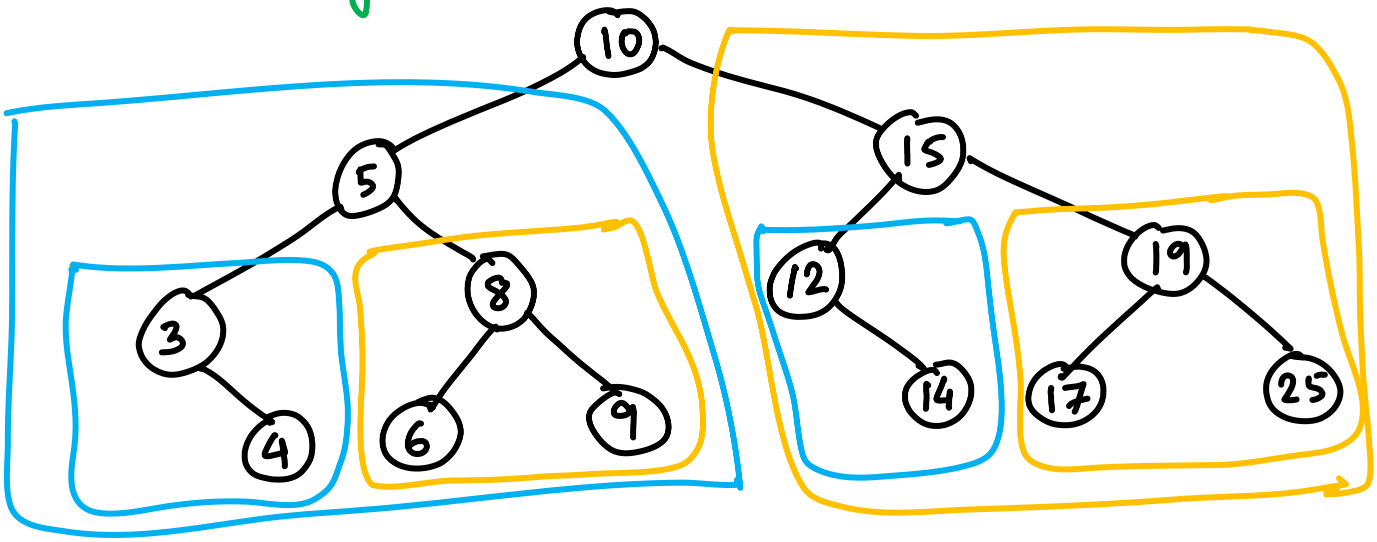
Information part within nodes of a "binary search tree" can be referred to as "Key"

A Binary Search Tree is such a binary tree which is either empty or which follows the below mentioned criteria :

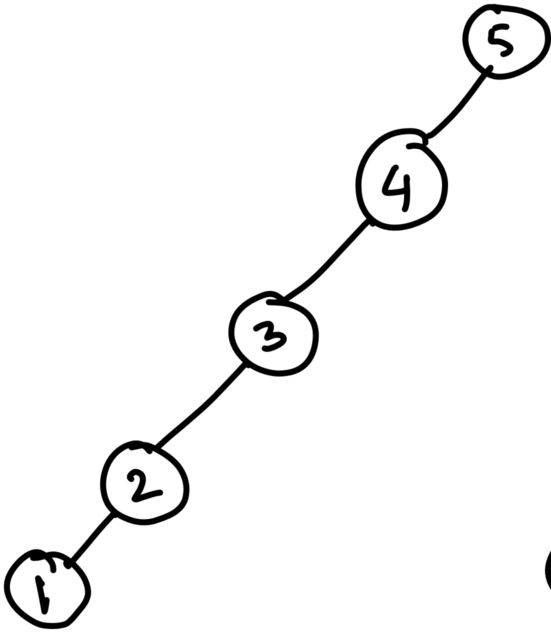
(1) all keys of the left-subtree of the root are less than the key in the root

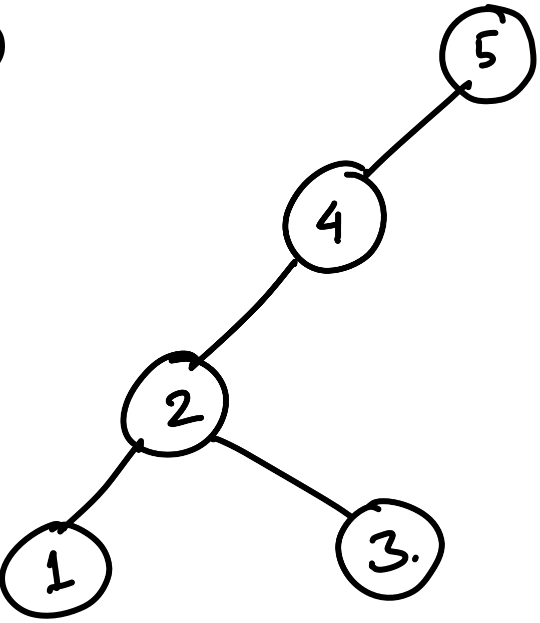
(2) all keys of the right-subtree of the root are greater than the key in the root.

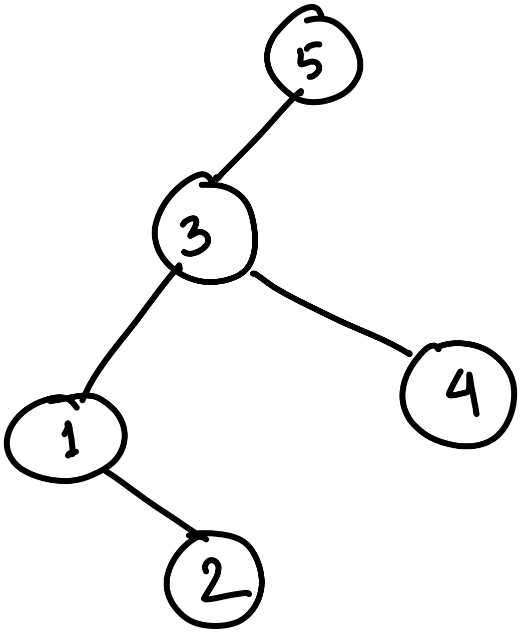
(3) the left and the right sub-trees of a binary search tree are binary search trees individually.

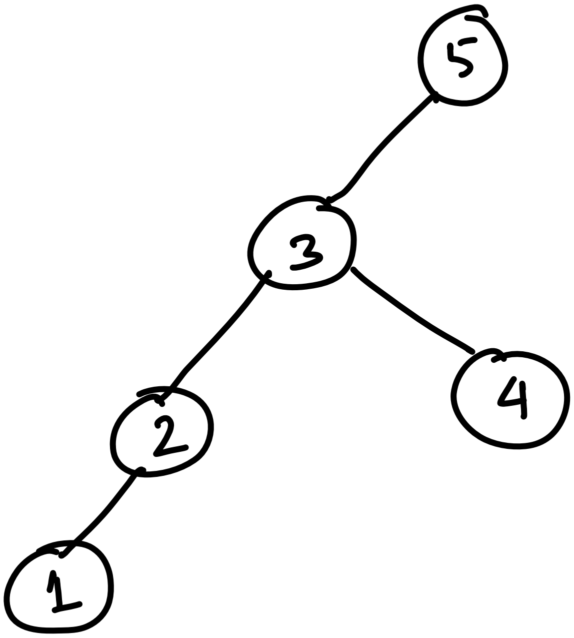


$S = \{1, 2, 3, 4, 5\}$

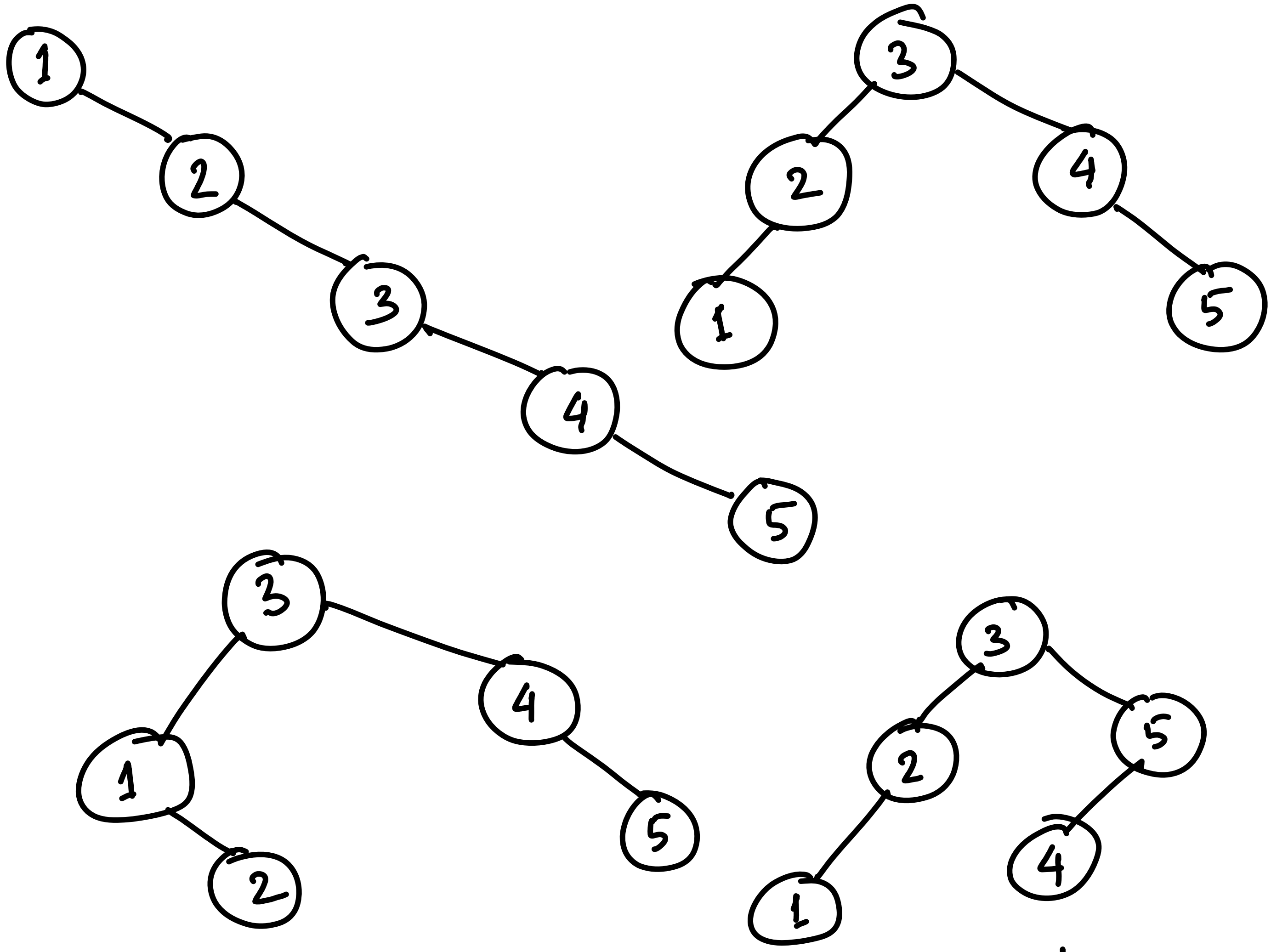








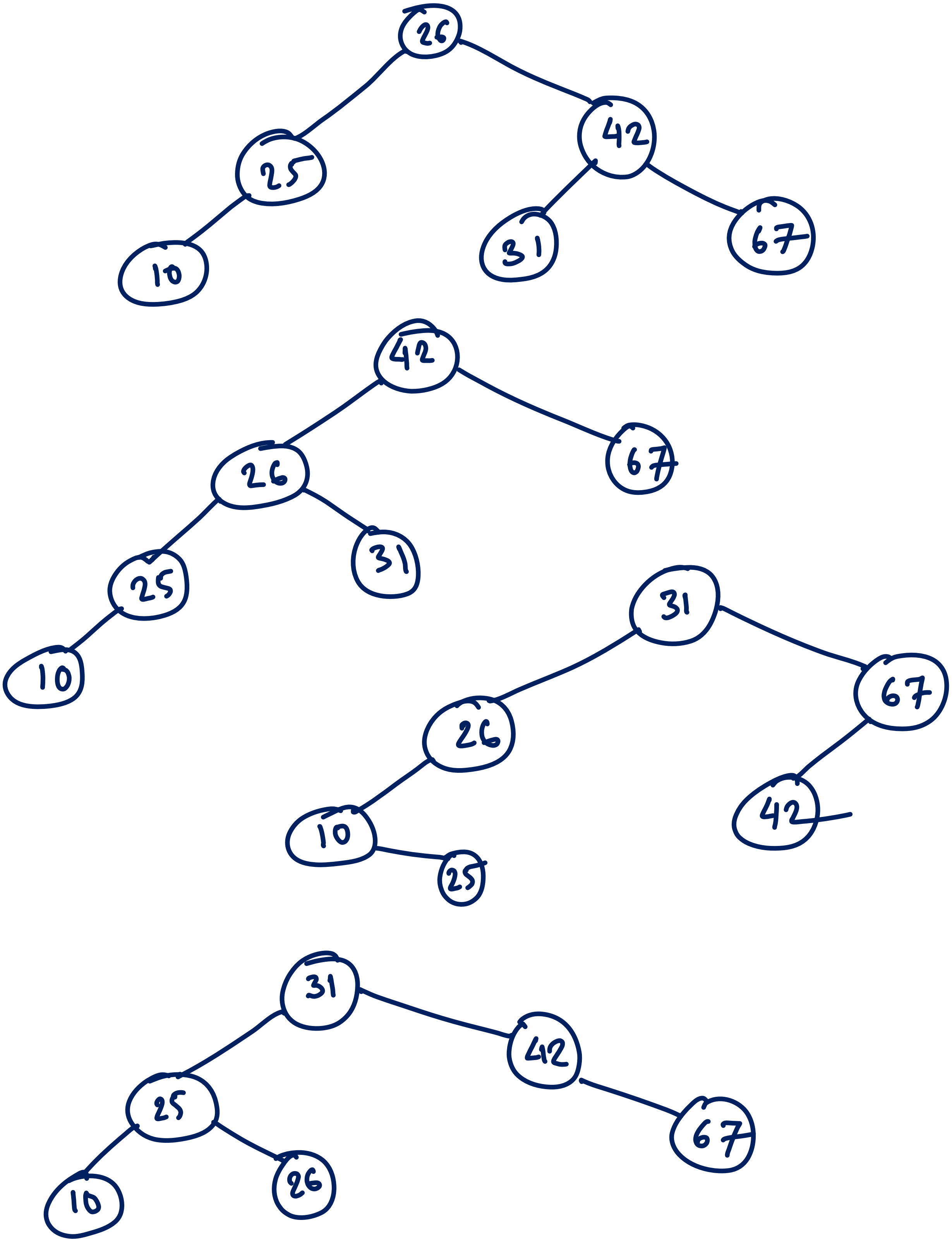
P.T.O



The above diagrams depit binary search tree formations for the set of key values, $S = \{1,2,3,4,5\}$

$S_1 = \{10, 25, 26, 31, 42, 67\}$

Draw binary search tree formations with the above key values within set S_1.



Searching within a Binary Search Tree (BST)

```
int search (struct node * root, int k)
  {
        if (root == NULL)
                return 0 ;
        if (root -> data == k)
                return 1 ;
        if (k < root -> data)
              search (root -> lchild , k) ;

        else
              search (root -> rchild , k) ;

    }
```

Creation of a BST :

```
      struct node
          {
                int data ;
                struct node * lchild ;
                struct node * rchild ;

            } ;
  struct node * root , * newnode ;
      :
```

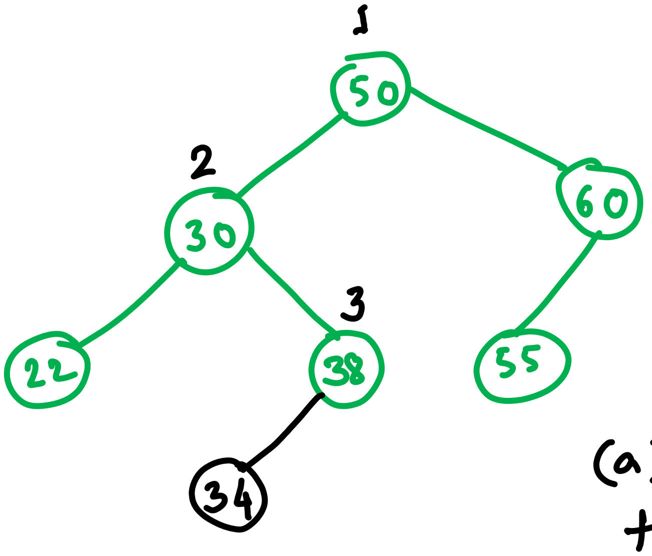
```
newnode = (struct node *) malloc (sizeof (struct node));
```

Insertion with a BST :

```
void insert_BST (struct node *root, struct node * newnode)
{   if (root -> data > newnode -> data)
      {   if (root -> lchild = = NULL)
                root -> lchild = newnode ;

          else
                insert_BST (root -> lchild, newnode);

      }
    else
      {   if (root -> rchild = = NULL)
                root -> rchild = newnode ;

          else
                insert_BST (root -> rchild, newnode);

      }
}
```

Searching & Insertion within a Binary Search Tree (BST)

dated on 22|12|2021



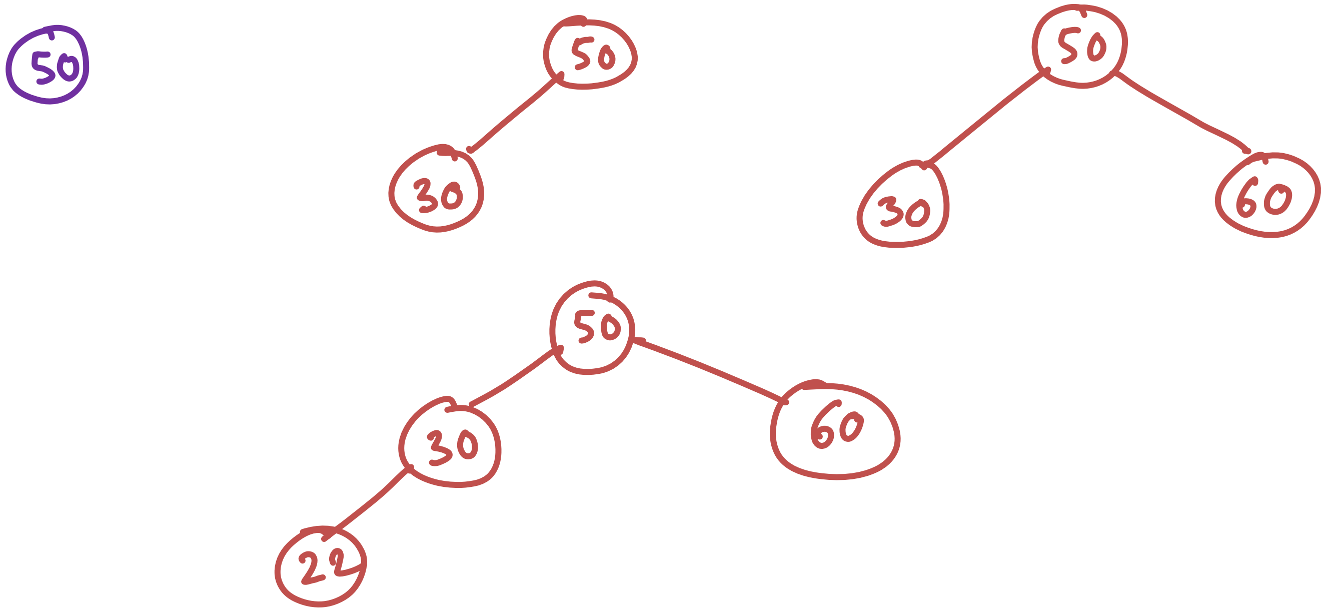
Say, The newnode which is to be inserted has a data value of 34.

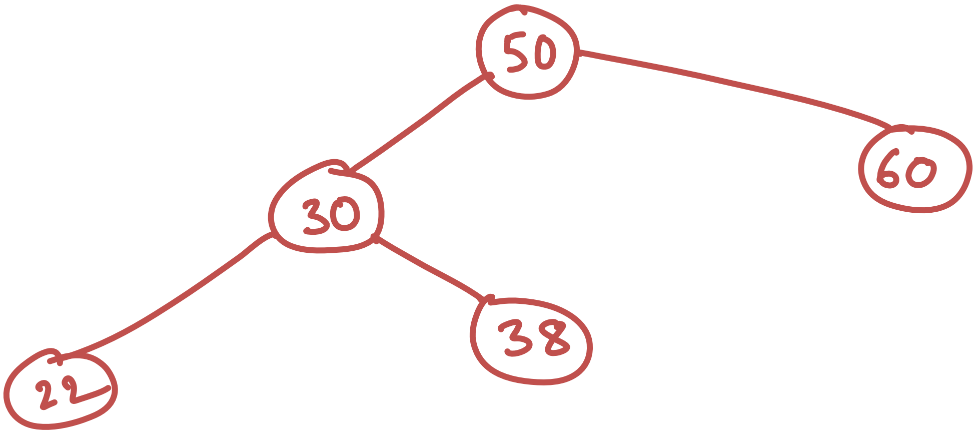
(a) if data < data of the root then compare it with the left child of the root

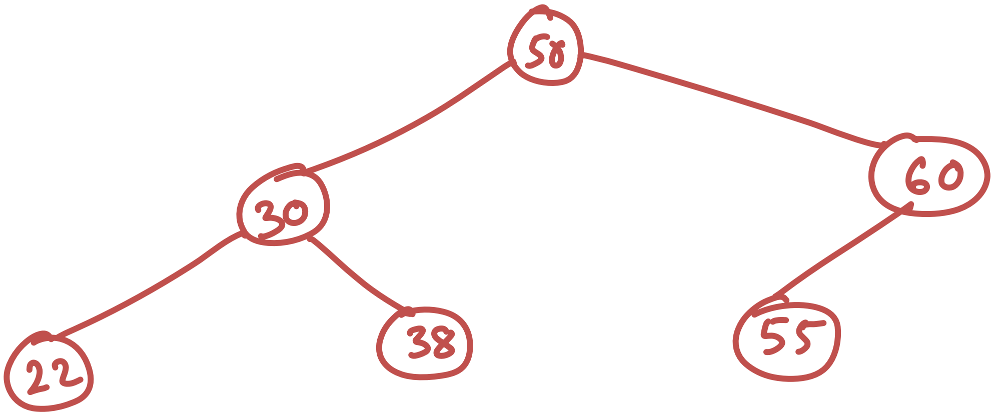
(b) if data > data of the root, then compare it with the right child of the root.

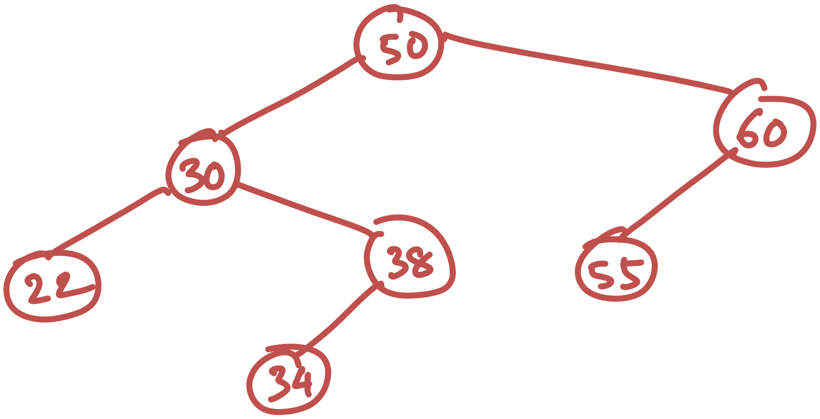
Let us take, seven numbers and insert them in a BST which is initially empty.

50, 30, 60, 22, 38, 55, 34



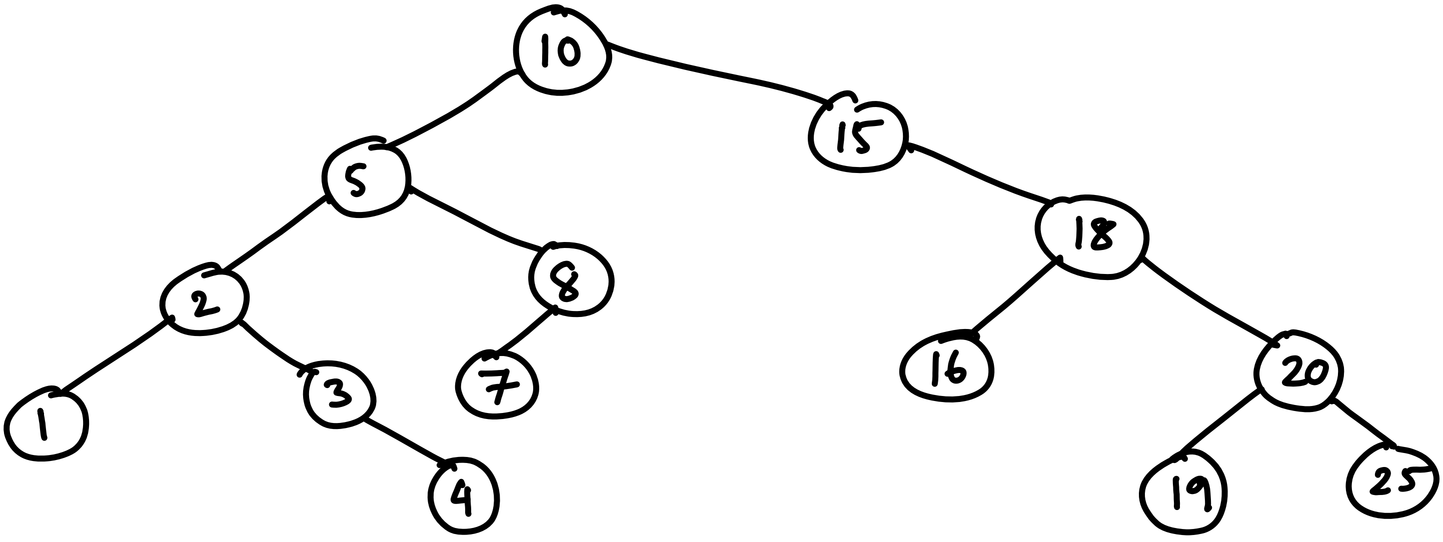






P.T.O

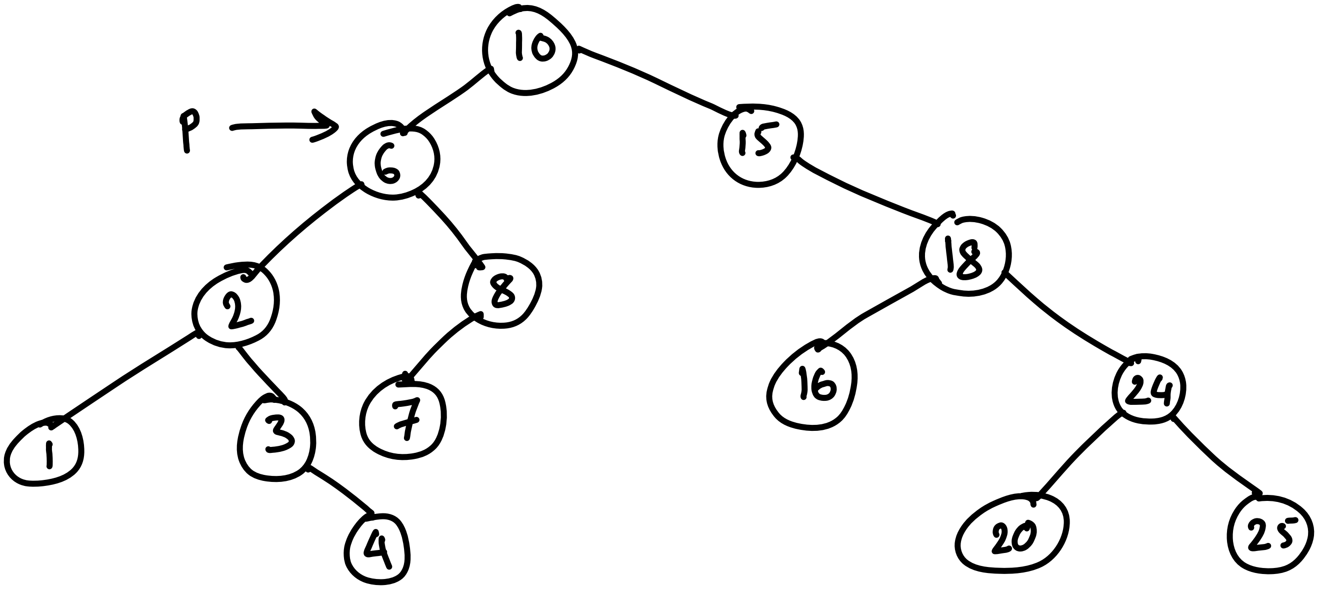
Deletion from a Binary Search Tree (BST)



(i) if the node to be deleted is a leaf node

(ii) if the node to be deleted has only one child

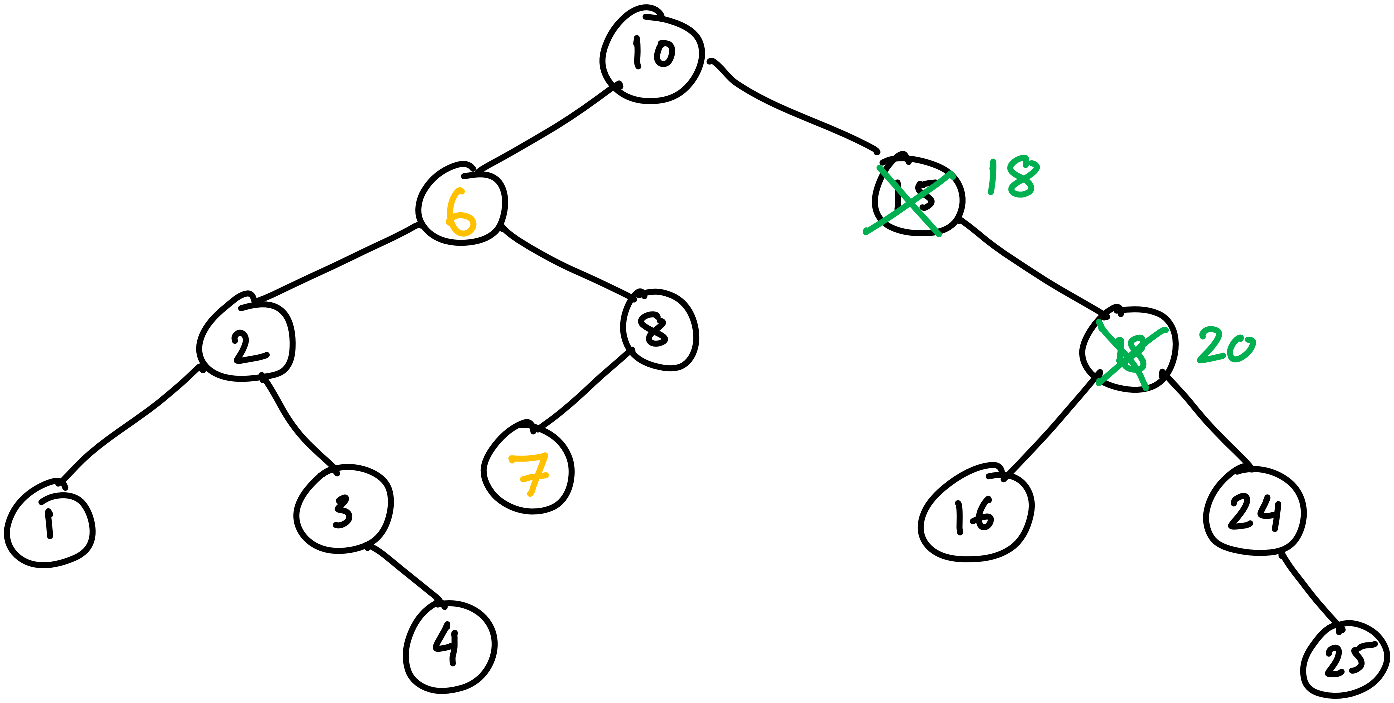
(iii) if the node to be deleted has two children.



Now, the node to be deleted must be replaced by its inorder-successor.

How to find the inorder successor of a node with 2-children within a BST ?

firstly, move to its right child and then Traverse continously to its left till we find a node with empty left child.



— x —

Deletion of a node within a Binary Search Tree (BST) [Contnd.]

Dated on 23/12/2021

```
struct node
        {
              int data ;
              struct node * left ;
              struct node * right ;
        } ;

struct node * Delete_BST(struct node * root, struct node * n
                                        struct node * parent)
{
    struct node * inorder_s ;
    struct node * x ;
    if ((n->left != NULL) && (n->right != NULL))
                            /* The 1st "if" block when the
       {                       to be deleted node has 2 children */
            parent = n ;
            inorder_s = n->right ;
            while (inorder_s->left != NULL)
                {     parent = inorder_s ;
                      inorder_s = inorder_s->left ;
                }
            n->data = inorder_s->data ;
            n = inorder_s ;
       }
if (n->left == NULL)
 {   if (parent) /* shall be true if prent != NULL */
      {     if (parent->left == n)
                 parent->left = n->right;
            else
                 parent->right = n->right;
            x = root; /* original root address
                           is retain */
      }
     else
          x = n->right; /* The right child of the
                            root is now the root of
                            the BST */
 }
if (n->right == NULL)   /* This is 3rd "if" block */
 {   if (paent)
      {     if (paent->left == n)
                 parent->left = n->left;
            else
                 parent->right = n->left;
            x = root; /* The original root address
                          is retained */
      }
     else
          x = n->left; /* The left child of
                           the root is now the
                           root of the BST */
 }
return x; /* it returns the root address of the
               finally modified BST */
}
```

Here, within the above deletion function, there are three "if" blocks.

The first "if" block gets executed when the node-to-be-deleted has both left child and right child.

The second "if" block gets executed when the node-to-be-deleted is a leaf node or it has only a right child.

The third "if" block gets executed when the node-to-be-deleted is a leaf node or it has only a left child.

It must be mentioned that if the node-to-be-deleted has both its children and its inorder successor is a leaf node, then also the other two "if" blocks shall be visited. But, if the inorder successor is not a leaf node, then the 2nd "if" block shall also be visited (not the 3rd one).

—— x ——