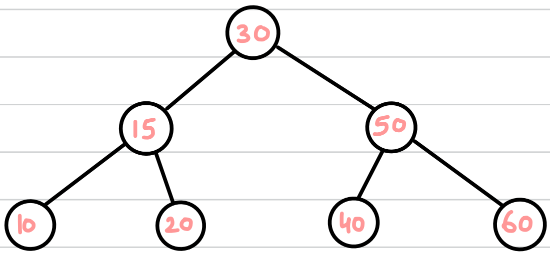
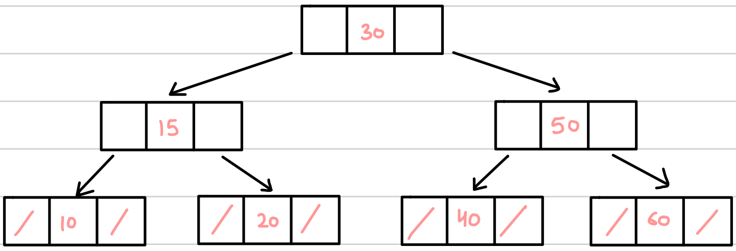
BINARY SEARCH TREE



- Left subtree is smaller
- Right subtree is greater
- Useful for searching (in less number of comparisons)
- No duplicates
- Inorder gives sorted order
- For n number of nodes, catalan number of trees can be generated

$$T(n) = \frac{{}^{2n}C_n}{n+1}$$



- BST is represented using linked representation.
- Can also be represented using arrays.

SEARCHING IN A BINARY SEARCH TREE

(searching takes maximum time same as height of tree) O(logn)

(1) RECURSIVE SEARCH (tail recursion)

O(h)

$\log n \le h \le n$

```
Node* Rsearch(Node *t, int key)
{
    if (t == NULL)              // If element is not
        return NULL;            found
    if (key == t→data)
        return t;
    else if (key < t→data)
        return Rsearch(t→lchild, key);
    else
        return Rsearch(t→rchild, key);
}
```

(2) ITERATIVE VERSION O(log n)

```
Node * Search(Node *t, int key)
{
    while (t != NULL)
    {
        if (key == t→data)
            return t;
        else if (key < t→data)
            t = t→lchild;
        else
            t = t→rchild;
    }
    return NULL;
}
```

INSERTING IN BINARY SEARCH TREE

```
void Insert(Node *t, int key)
{
    Node *r = NULL, *p;

    while(t != NULL)
    {
        if(key == t->data)
            return;        // To avoid duplication
        else if(key < t->data)
            t = t->lchild;
        else
            t = t->rchild;
    }
    p = new Node;
    p->data = key;
    p->lchild = p->rchild = NULL;

    if(p->data < r->data)
        r->lchild = p;
    else
        r->rchild = p;
}
```

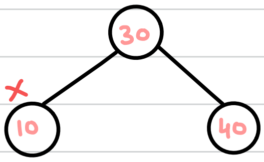
RECURSIVE INSERT IN BST

```
Node* Insert(Node *p, int key)
{
    if(p == NULL)
    {
        t = new Node;
        t->data = key;
        t->lchild = t->rchild = NULL;
    }
    if(key < p->data)
        p->lchild = insert(p->lchild, key);
    else if(key > p->data)
        p->rchild = insert(p->rchild, key);
    return p;
}
```

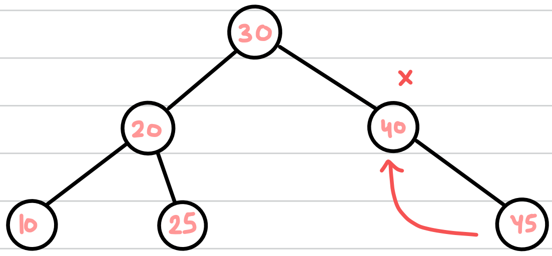
```
void main()
{
    Node *root = NULL;
    root = insert(root, 30);
    insert(root, 20);
    insert(root, 25);
}
```

DELETING FROM BST

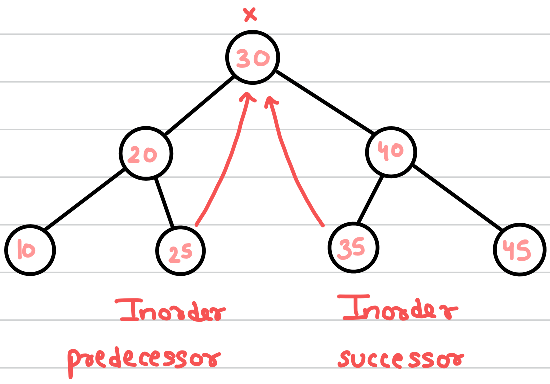
CASE 1 :



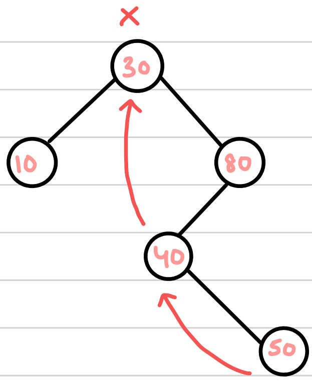
CASE 2 :



CASE 3 :



CASE 4 :



// Multiple changes to be made

```
int Height (struct Node *p)
{
        int x, y;
        if (p == NULL)
                return 0;
        x = Height (p -> lchild);
        y = Height (p -> rchild);
        return x>y ? x+1 : y+1;
}
```

```
Struct Node * InPre (struct Node *p)
{
        while (p && p -> rchild != NULL)
                p = p -> rchild;
        return p;
}
```

```
struct Node *InSucc (Struct Node *p)
{
        while (p && p -> lchild != NULL)
                p = p -> lchild;
        return p;
}
```

```
struct Node * Delete(struct Node *p, int key)
{
    struct Node *q;

    if (p == NULL)
        return NULL;

    if (p → lchild = NULL && p → rchild = NULL)   // Leaf Node
    {
        if (p == root)
            root = NULL;
        free(p);
        return NULL;
    }

    if (key < p → data)
        p → lchild = Delete(p → lchild, key);
    else if (key > p → data)
        p → rchild = Delete(p → rchild, key);
    else
    {
        if (Height(p → lchild) > Height(p → rchild))
        {
            q = InPre(p → lchild);
            p → data = q → data;
            p → lchild = Delete(p → lchild, q → data);
        }

        else
        {
            q = InSucc(p → rchild);
            p → data = q → data;
            p → rchild = Delete(p → rchild, q → data);
        }
    }
    return p;
}
```

GENERATING BINARY SEARCH TREE

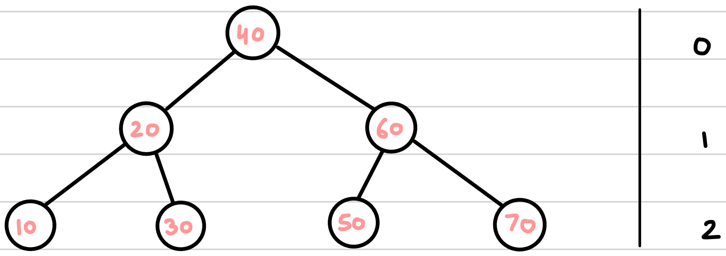
```
void Createpre( int pre[], int n)    O(n)
{
        stack st;
        Node *t;
        int i=0;

        root = new Node;
        root → data = pre[i++];
        root → lchild = root → rchild = NULL;
        p = root;

        while (i<n)
        {
                if (pre[i] < p → data)
                {
                        t= new Node;
                        t → data = pre[i++];
                        t → lchild = t → rchild = NULL;
                        p → lchild = t;
                        push(&stk, p);
                        p = t;
                }
                else
                {
                        if( pre[i] > p → data && pre[i] < stacktop(stk) → data)
                        {
                                t = new Node;
                                t → data = pre[i++];
                                t → lchild = t → rchild = NULL;
                                p → rchild = t;
                                p = t;
                        }
                        else
                        {
                                p = pop(&stk);
                        }
                }
        }
}
```

BST can be generated using
- Preorder + Inorder
- Postorder + Inorder

Sorted preorder of BST gives its Inorder /postorder

DRAWBACK OF BINARY SEARCH TREE

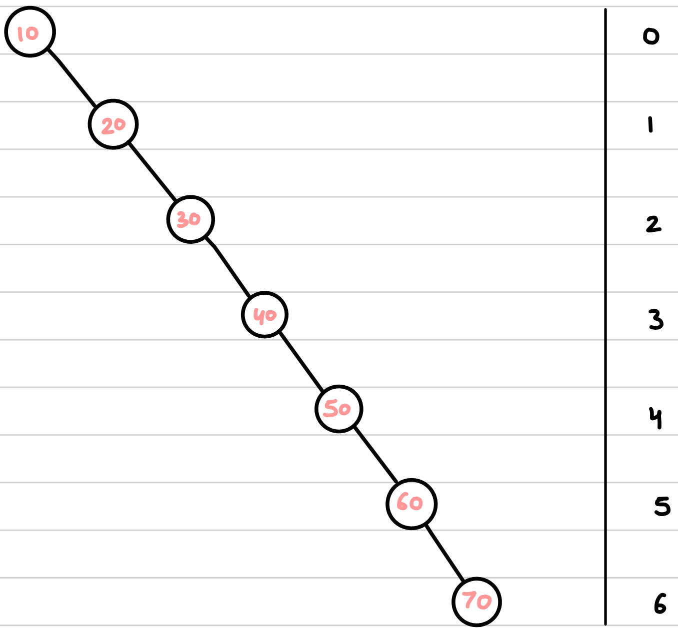
KEYS : 40, 20, 30, 60, 50, 10, 70



$h = \log_2(n+1) - 1$

$O(\log n)$

KEYS : 10, 20, 30, 40, 50, 60, 70



$h = n-1$

$O(n)$

There is no control over height of binary search tree.

It only depends on user's input or sequence of insertion