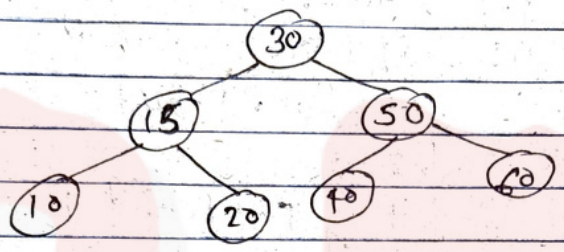
Binary Search Tree

In this tree, for every node, all the elements of its left subtree are smaller than that node and its right subtree are greater than that node.



Properties

i) No Duplicates
ii) Inorder gives sorted order - (10, 15, 20, 30, 40, 50, 60)
iii) If 'n' nodes are given then no. of BST are $\frac{2n_{C_n}}{n+1}$.

* Searching in a binary search tree -

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

* It depends upon hight. (Here we consider min~height)

Time - O(log n)

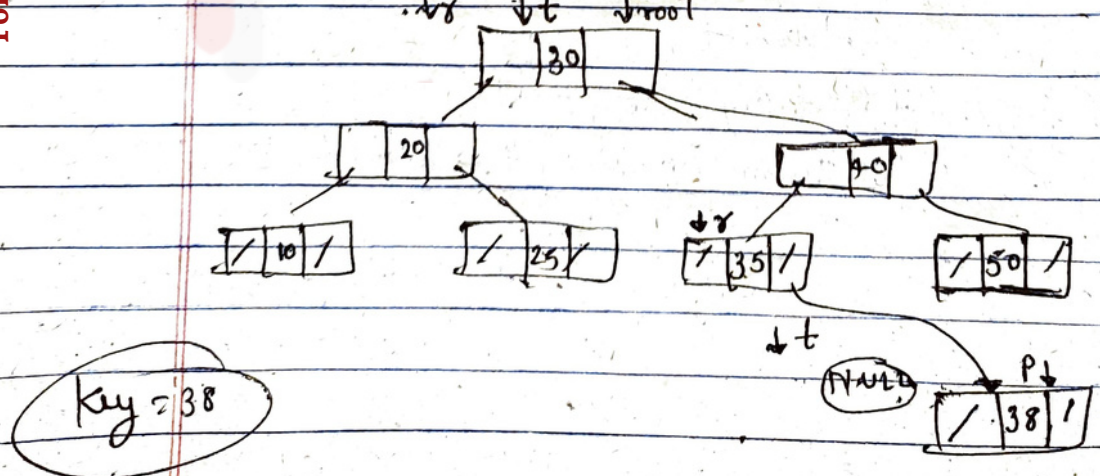
Iterative

```
Node * search (Node *t, int key)
{
    while (t != NULL)
    {
        if (key == t->data)
            return t;
        elseif (key < t-> data)
            t = t->lchild;
        else
            t = t->rchild;
    }
}
```

* Inserting in a binary search tree -

i) Check or search that key which is inserted in the tree, is already present or not. If already present then we can not insert it in the tree.

ii) Here we use two pointers, first is used for checking the element and second is used as tailing pointer.



```
void Insert (Node *t, int key)
{ Node *r = NULL, *P;
  while (t != NULL)
  { r = t;
    if (key == t->data)
      return;
    elseif (key < t->data)
      t = t->lchild;
    else
      t = t->rchild;
  }
  P = malloc (- -);
  p->data = key;
  P->lchild = P->rchild = NULL;
  if (P->data < r->data)
    r->lchild = P
  else
    r->rchild = P;
}
```

time - O(log n)

t → 't' is firstly pointing on root node then it will move for checking the suitable position where 'key' can be inserted and it is also check for duplicate key.

P → (M) For making new node

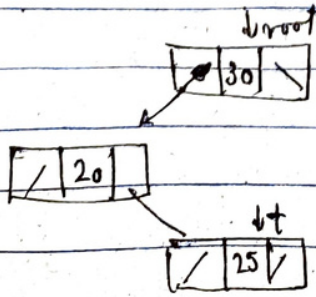
r → Trailing pointer and it is also check for duplicate key.

* Recursive insert in Binary Search tree

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

time
O(n log n)

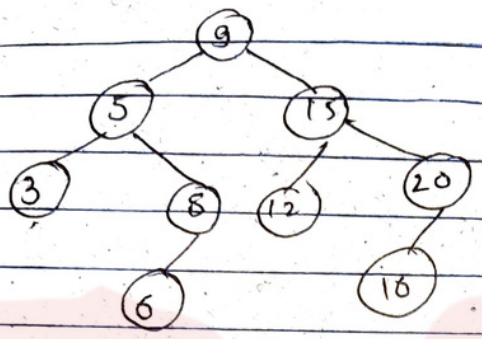
```
int main()
{   Node *root = NULL;
    root = insert (root, 30);
        insert (root, 20);
        insert (root, 25);
}
```



Creating a Binary Search Tree

Eg:- keys - 9, 15, 5, 20, 16, 8, 12, 3, 6

Date
Page



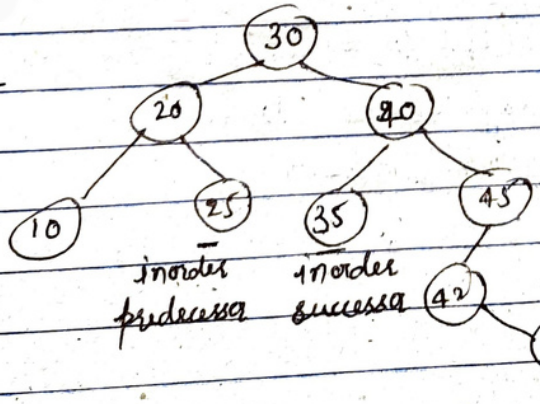
inserting - n
Searching - log n
time - n × log n
= O(n log n)

* Deleting from Binary Search Tree - (value not node)
When we delete an element of a tree then it's place is taken by its (element which is deleted) inorder predecessor or inorder successor.

→ Inorder predecessor -
go to left sub-tree of deleted node then right:-right--right. (right most child).

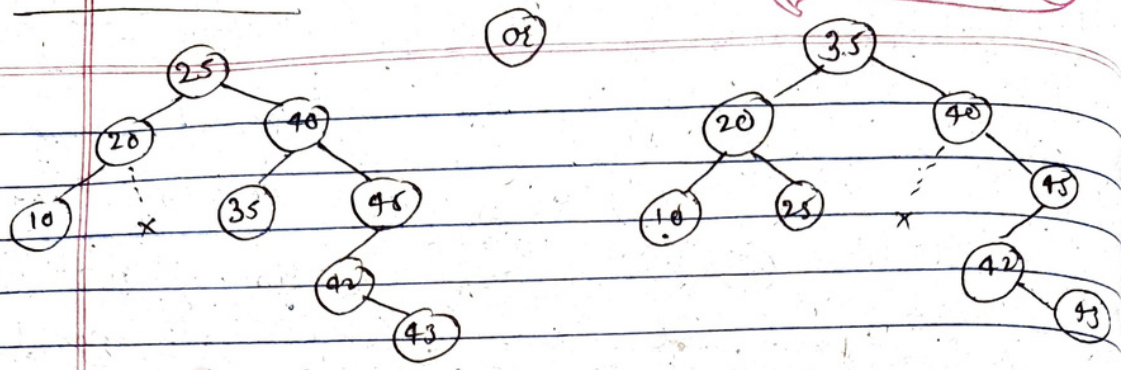
→ Inorder Successor:
~~go to right sub-tree~~
Left most child of right sub-tree.

Eg:-



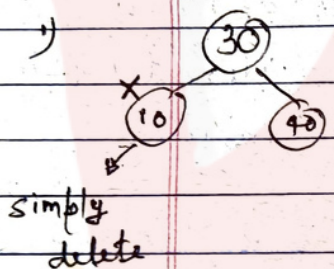
→ Here, If we want to delete 30 then its place is taken by either inorder predecessor (25) or ~~and~~ inorder successor (35).

After deleting



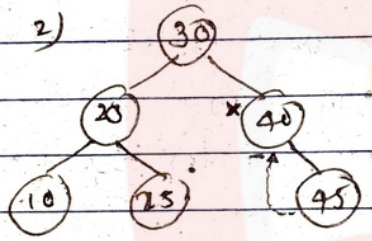
* Different cases of Deleting an element in BST

1)

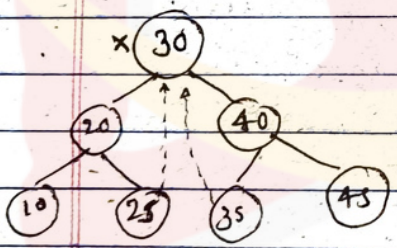


simply delete

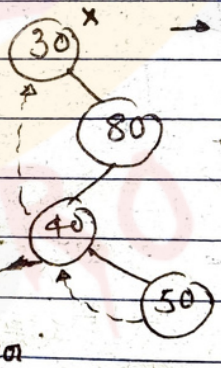
2)



3)



4)



→ Here more than one modification are done

it is inorder successor for 30.

* When we are deleting a node from the BST, either it may be a leaf node or it will have exactly one child.