Successor and Predecessor[1]

```
TREE-SUCCESSOR (x)
    If Right [x] ≠ NIL
        then return TREE-MINIMUM (Right [x])
        y ← P[x]
    while Y ≠ NIL and x = Right [Y]
        do x ← y
            y ← P(y)
    return y.
```

If right subtree of node x is nonempty, then the successor of x is just the left most node in the right subtree. Otherwise if right subtree is empty and x has a successor y, then y is lowest ancestor of x whose left child is also an ancestor of x.

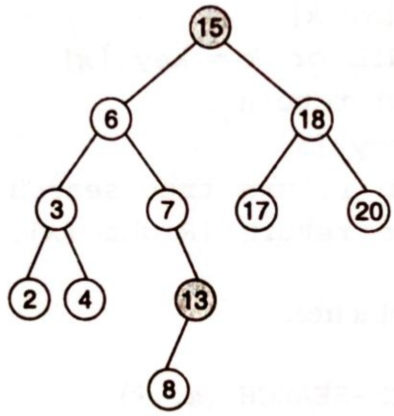


Fig. 13.6. Binary Tree.

As shown in Fig. 13.6 above successor of node with key 13 is the node with key 15. To find y, simply go up the tree from x until we encounter a node that is left child of its parent.

Running time of TREE-SUCCESSOR on a tree of *ht.* h is $O(h)$ Similarly we can have PREDECESSOR algorithm.

Insertion and Deletion

```
TREE-INSERT (T, Z)
    y ← NIL
    x ← Root [T]
while x ≠ NIL
    do y ← x
        if key [Z] < key[x]
            then x ← Left [x]
            else x ← Right [x]
P[z] ← y
```

1 Successor of a node x is the node with the smallest key greater than key $[x]$.

```
if y = NIL
    then Root [T] ← [Z]
    else if key [Z] < key [Y]
    then Left [y] ← Z
    else Right [Y] ← Z
```

Tree-Insert begins at the root of the tree and traces a path downward. The pointer x traces the path and pointer y is maintained as the parent of x. The two pointers move down comparing key [z] with key [x] until x is set to NIL. At this position Z is inserted.

key [z] = 13.

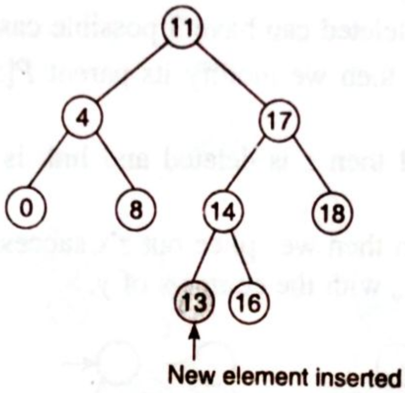


Fig. 13.7. Shows insertion.

key [x] = 11.

$13 \nless 11$

$\therefore$ $x \leftarrow$ Right [x]

(*i.e.*, at key [x] = 17)

$13 < 17$.

$\therefore$ $x \leftarrow$ left [x]

(*i.e.*, key [x] = 14)

$13 \nless 14$.

$\therefore$ $x \leftarrow$ left [x] = NIL

So key [z] is inserted at this position.

Running time is $O(h)$

```
Tree-Delete (T, Z)
    If Left [z] = NIL or Right [z] = NIL
        then y ← z
        else y ← TREE-SUCESSOR (Z)
    if Left [y] ≠ NIL
        then x ← Left [y]

        else x ← Right [y]
```

```
if x ≠ NIL
      then P(x) ← P(y)
if P[y] = NIL
      then Root [T] ← x
      else if y = left [P [y]]
      then Left [P(y)] ← x
      else Right [P(y)] ← x
if y ≠ z
      then key [z] ← key [y]
return y
```

A node z which is to be deleted can have 3 possible cases.

(i) Node have no children then we modify its parent $P[z]$ to replace z with its NIL as its child.

(ii) Node have single child then z is deleted and link is set between its parent and child directly.

(iii) Node have two children then we splice out z's successor y, which has no left child and replace the contents of z with the contents of y.

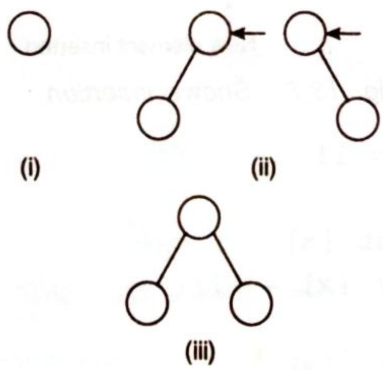
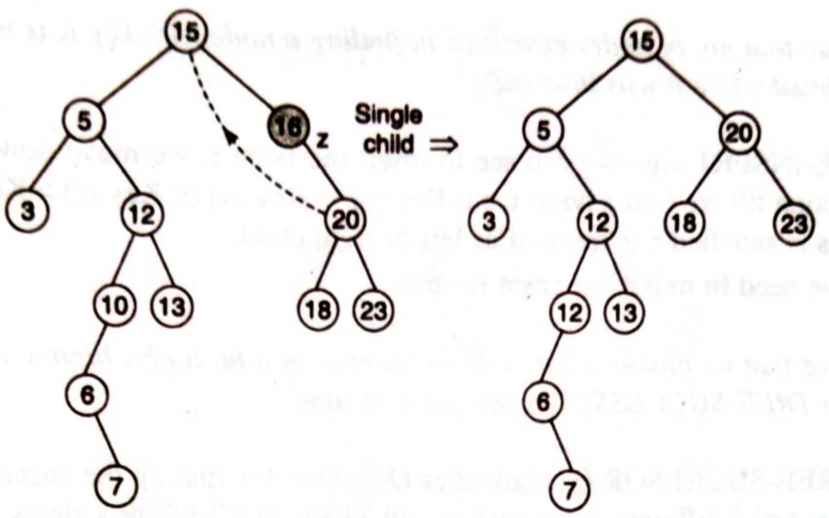


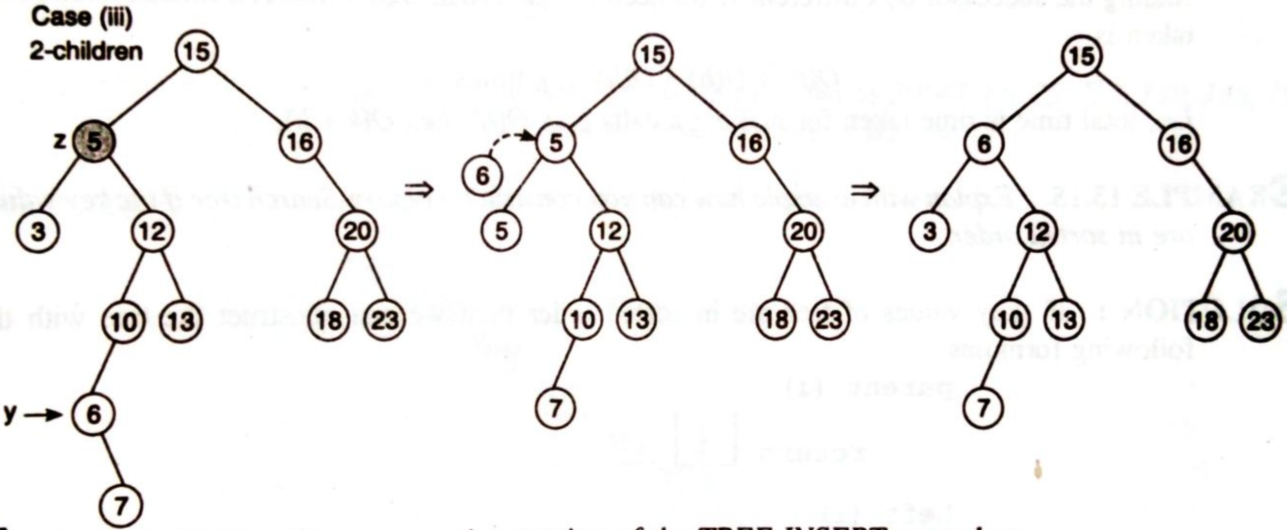
Fig. 13.8. Three cases for deletion.

Fig. 13.9 (a)–(c). Show BS tree deletion.

Case (ii)



Case (iii)
2-children



EXAMPLE 13.12. *Give a recursive version of the TREE-INSERT procedure.*

SOLUTION :

```
TREE-INSERT-REC (T, z, x)
        y ← x
        if (Key (z) < Key [x] and x ≠ NIL)
                    TREE-INSERT-REC (T, z, Left [x])
        else        TREE-INSERT-REC (T, z, Right [x])
                    P(z) ← y
                    if y = NIL
                    then root T ← z
                    else if Key [z] < Key [y]
                    then left [y] ← z
                    else Right [y] ← z
```