BINARY SEARCH TREE

Binary Search Tree (BST)

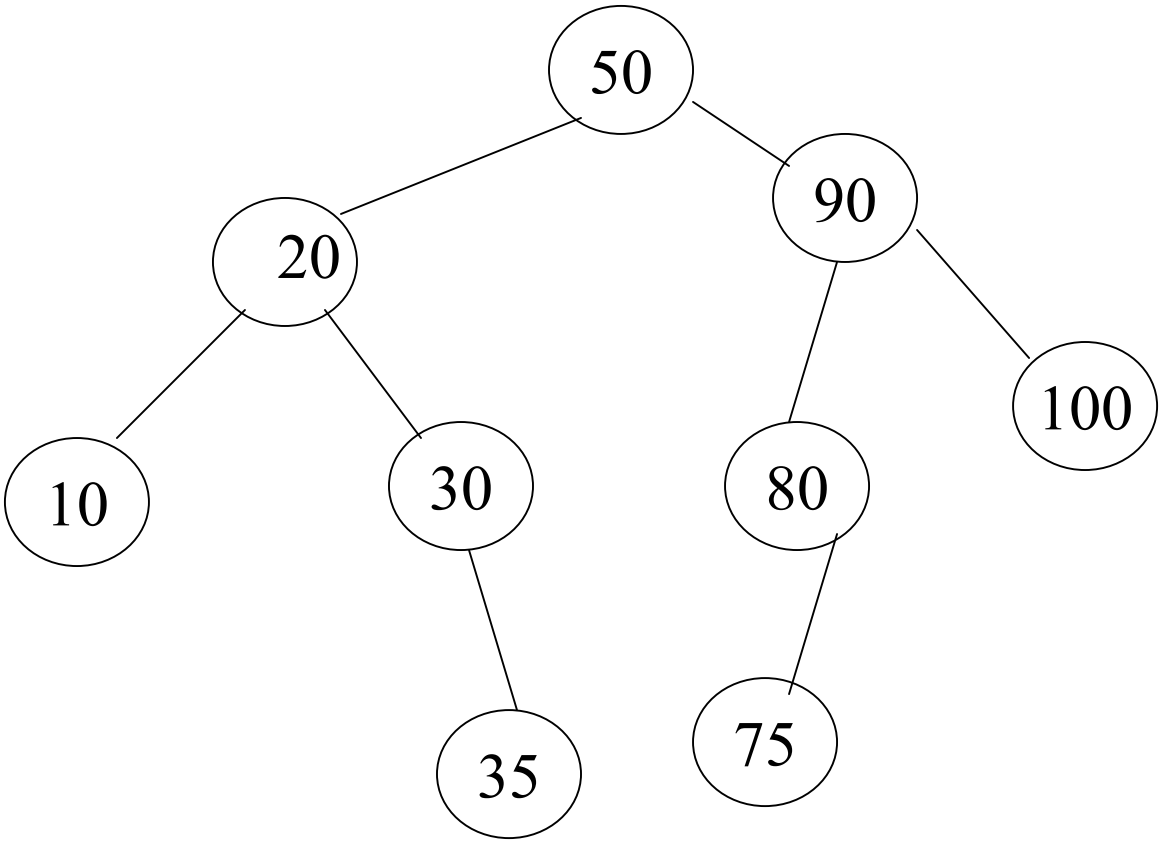
A BST is a binary tree T with the following conditions:

a) Key of every node in the right sub-tree of T is greater then the Key at root.

b) Key of every node in the left sub-tree of T is less then the Key at root .

c) All Keys are distinct.

An Example



BST Operations

1.Search for a key

2.Insert a key

3.Delete a key

4.Findmax & Findmin

5.Find the Kth max or min

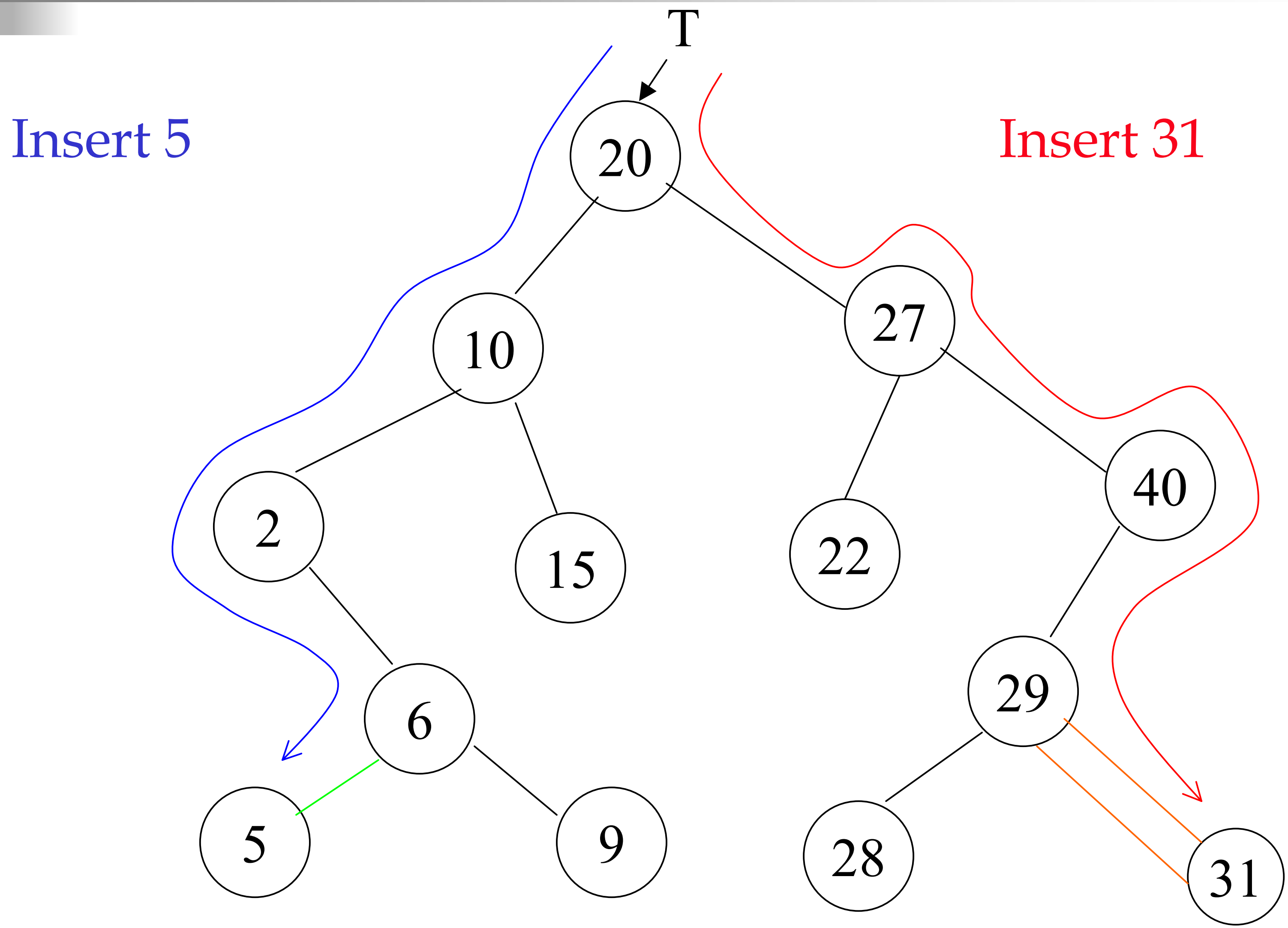
Recursive Search

```
BST * search (T key, BST * t){
  if (empty_t(t))
        return NULL;
  else if (key==t→info)
              return t;
        else if (key < t→info)
              return (search (key,t→left));
              else
                    return (search (key, t→right));
}
```

Non-recursive Search

```
BST * search (T key, BST * t) {
  BST *cur ; int found;
  if (empty_t(t))
      return NULL;
  else{
         cur=t; found=0;
          while( (cur!=NULL) & (!(found))){
              if (key==cur → info)  found=1;
                              else if (key < cur→info)
                                          cur=cur→left;
                              else cur=cur→right;
   }
  return cur;
 }  }
```

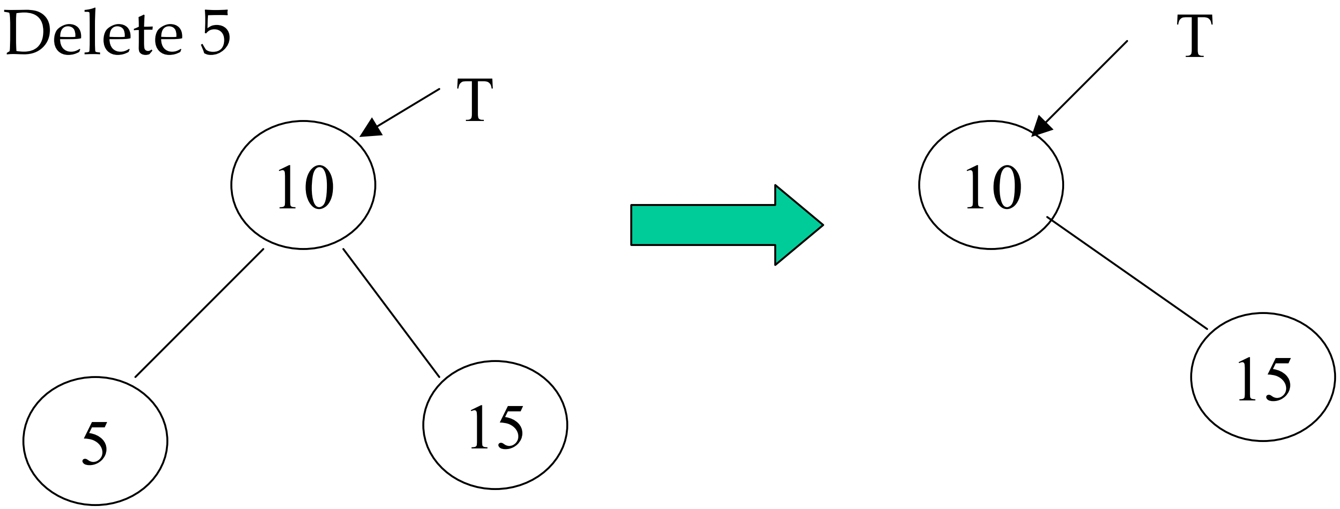
Insertion Example

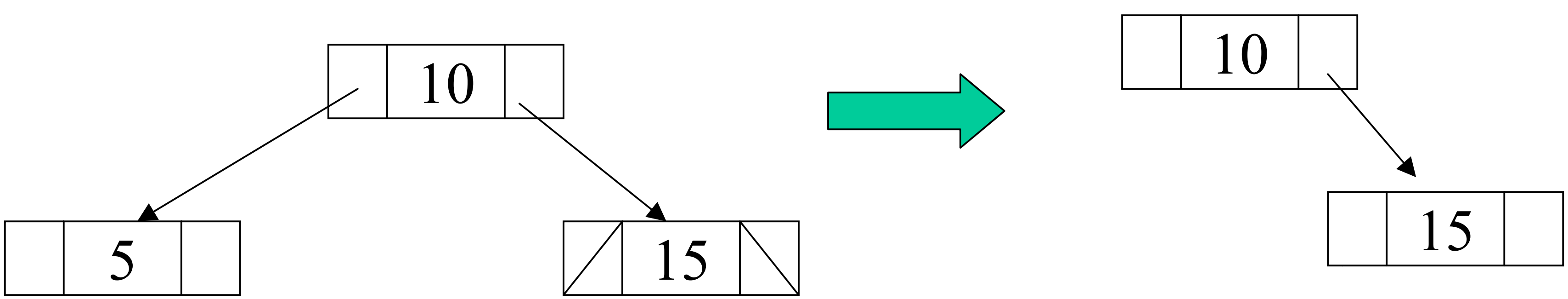


Deletion Example

- Delete 5

- Delete 5

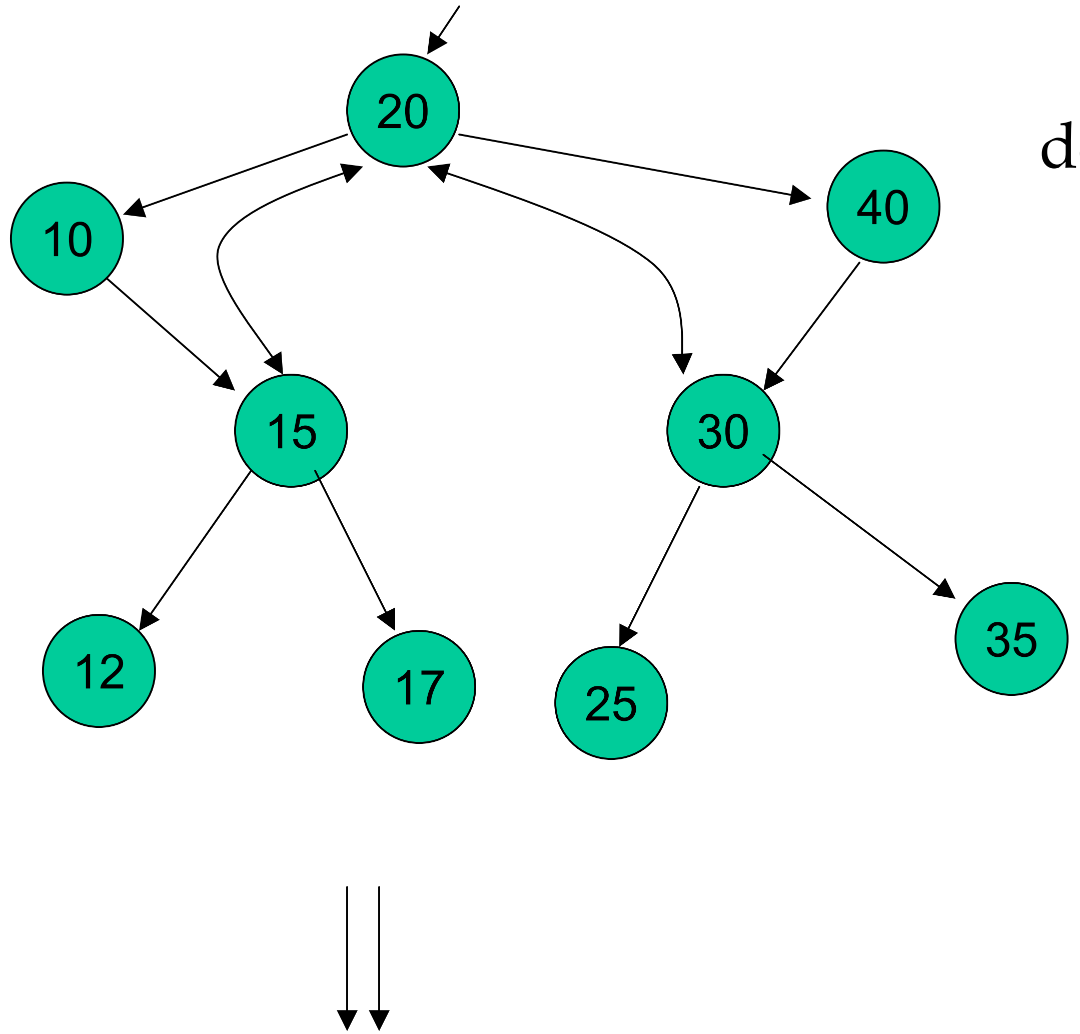




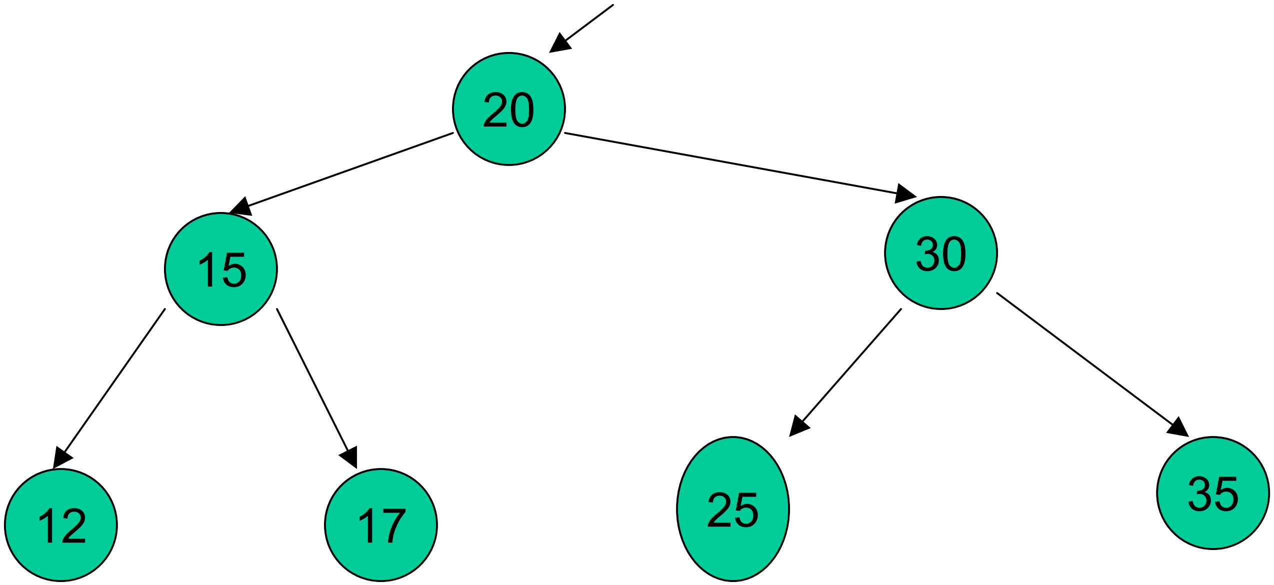
Deletion Example …

delete 10

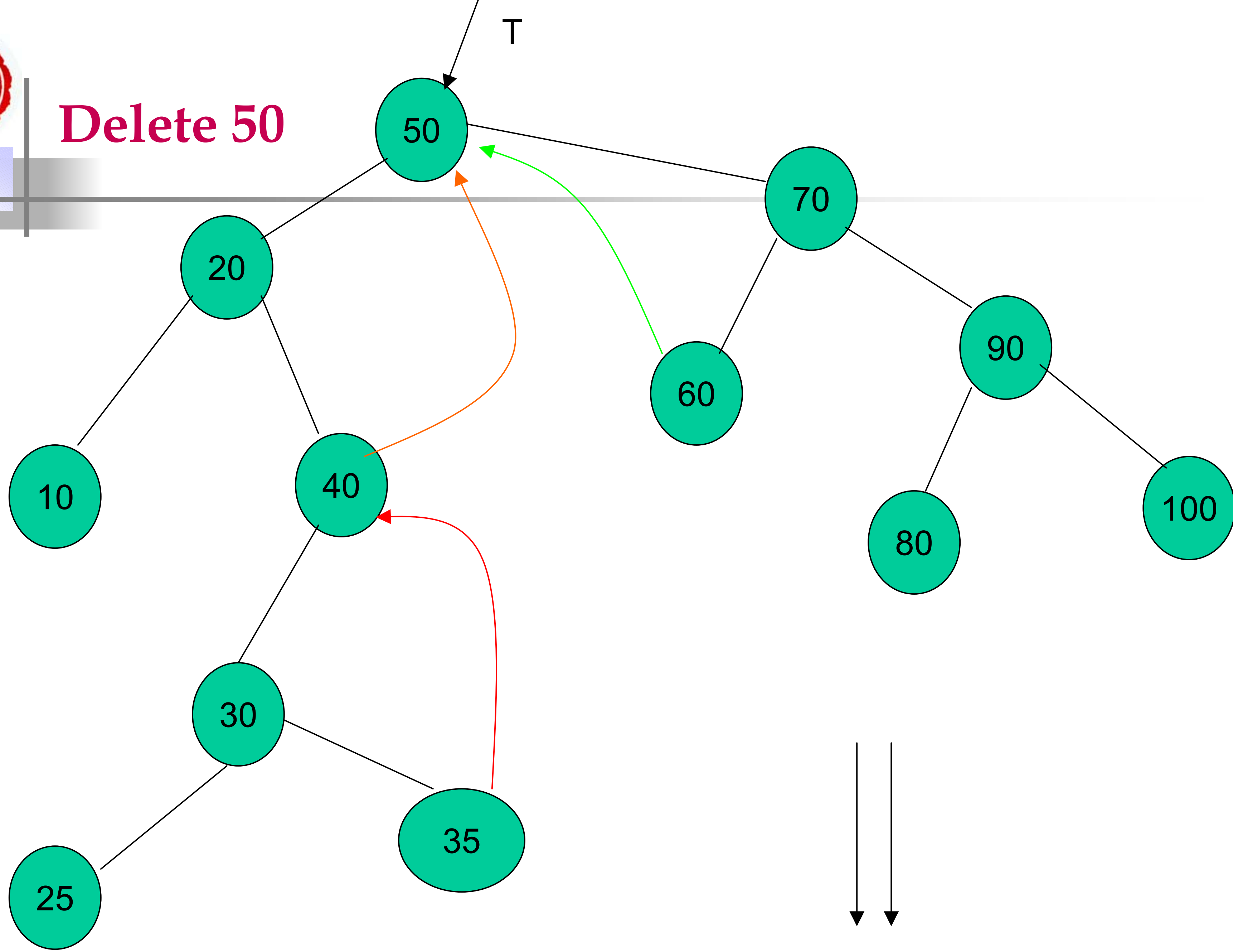
delete 40



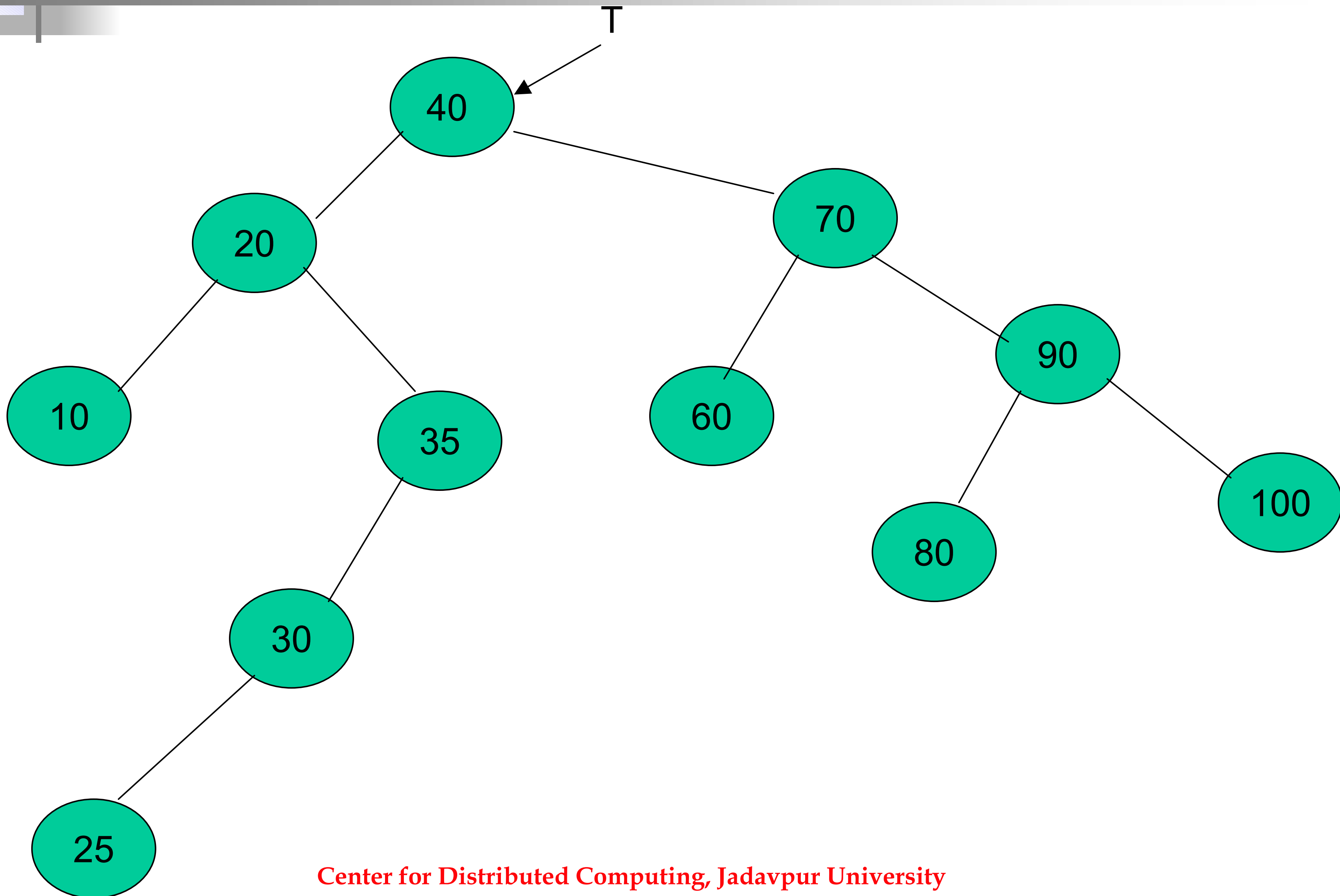
Deletion Example …



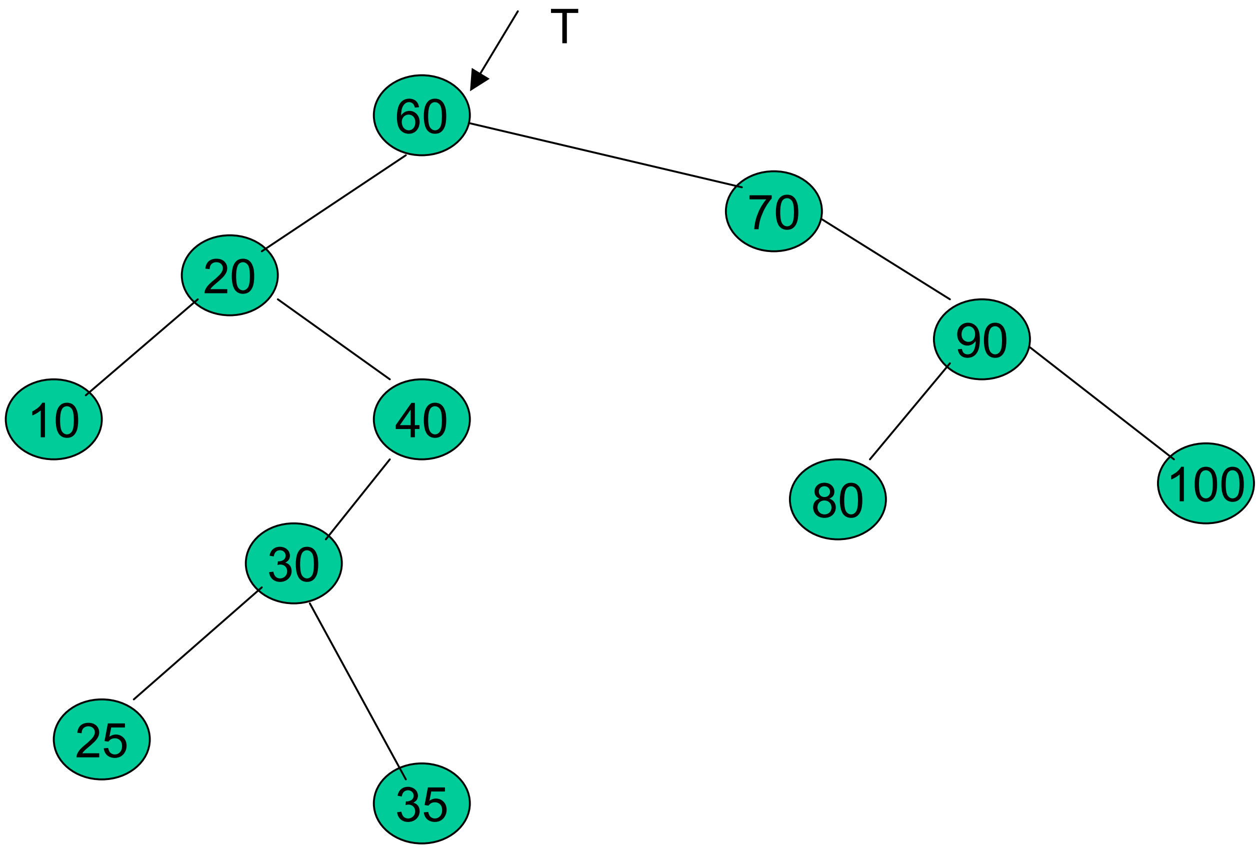
Delete 50



Result 1



Result 2



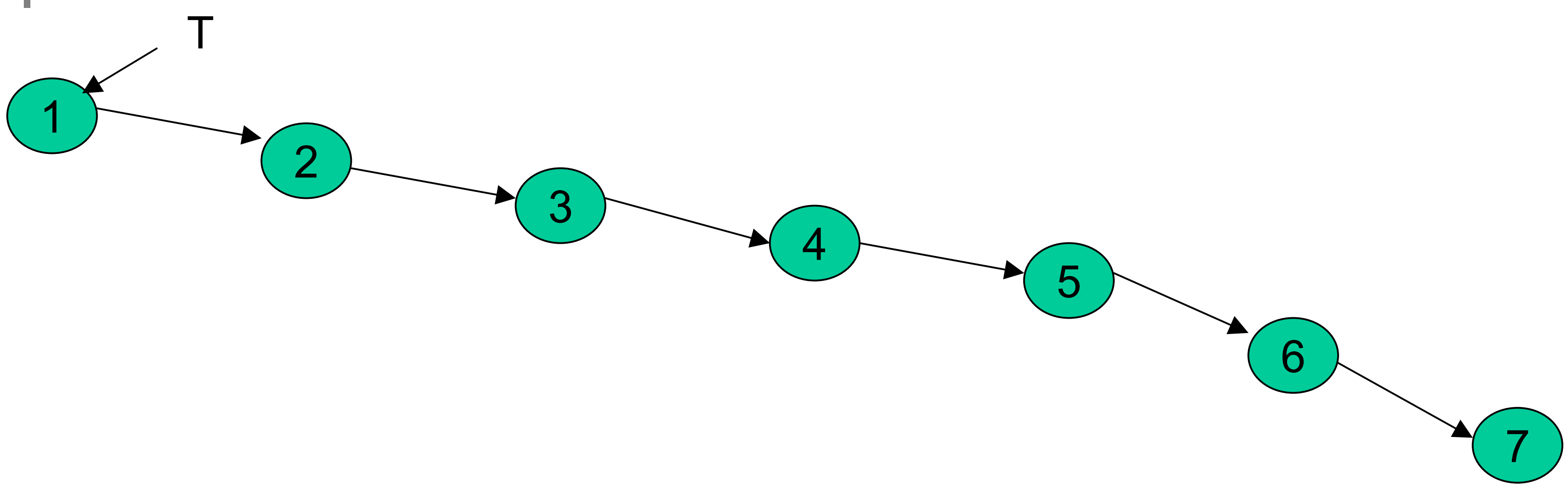
Problem of BST

- Average case complexity of search, insertion and deletion operations is $O(\log_2 n)$, where n is the no of nodes in the tree.
- The height of a BST depends on the sequence of insertion and deletion of keys.
- An extreme case:

Draw a BST for the following sequence of insertions:

1, 2, 3, 4, 5, 6, 7

Problems of BST …



The tree degenerates into a linked list.

The worst case complexity of search, insertion and deletion are O(n).

Remedy: Balanced tree.

Height Balanced Tree (AVL Tree)

- Invented by Adelson-Velskii, Landis

- AVL tree is a BST where at each node (including the root node) the left sub-tree and the right sub-tree do not differ in height by more than one.

$$|h_L - h_R| <= 1$$

Balance Factor

- Balance Factor (BF) of a node is the difference between the heights of its left and right sub-trees.

  $BF = h_L - h_R$

  BF = 1 left high
  BF = -1 right high
  BF = 0 equal high

AVL Tree Operations

1. Search a key

2. Find max & Find min

3. Find Kth max & kth min

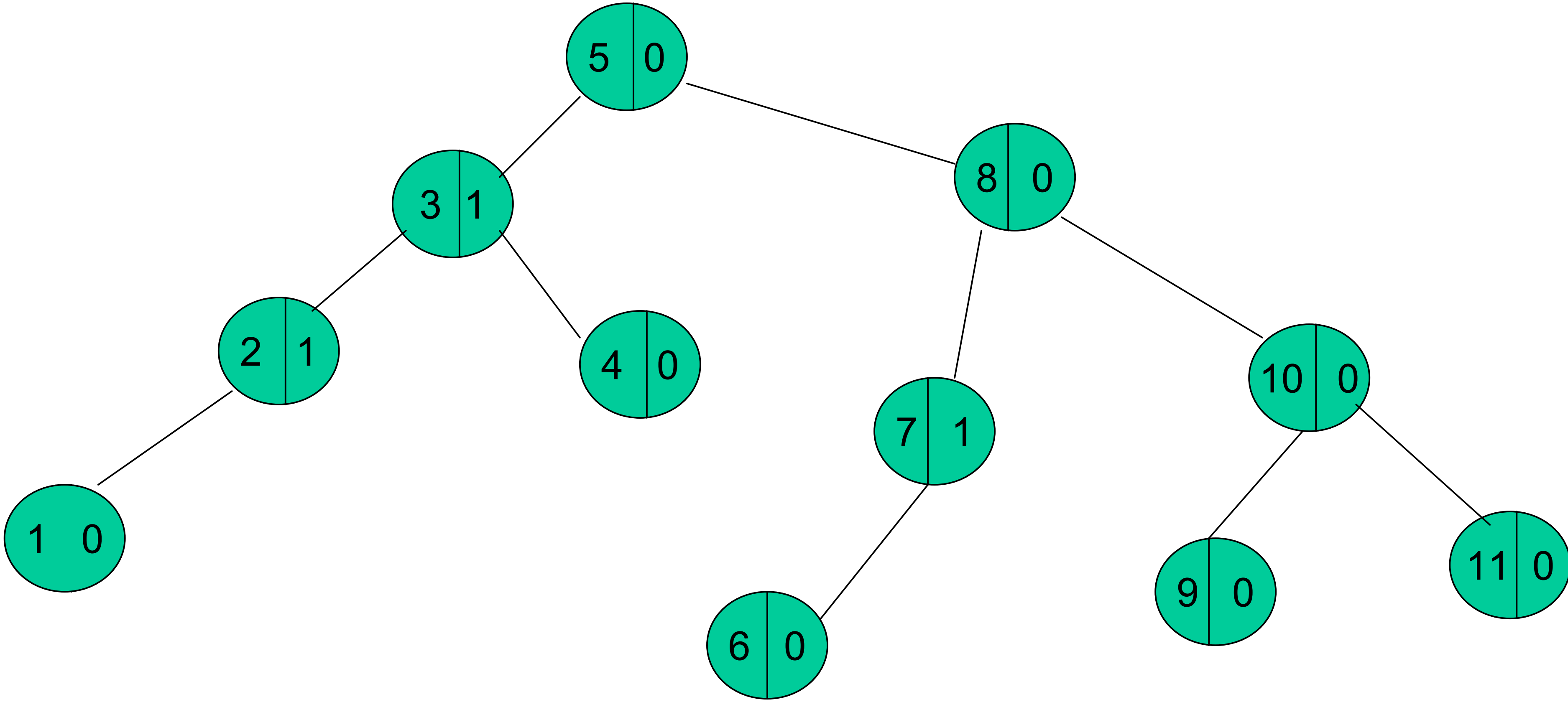
Same as BST

4. Insert a Key

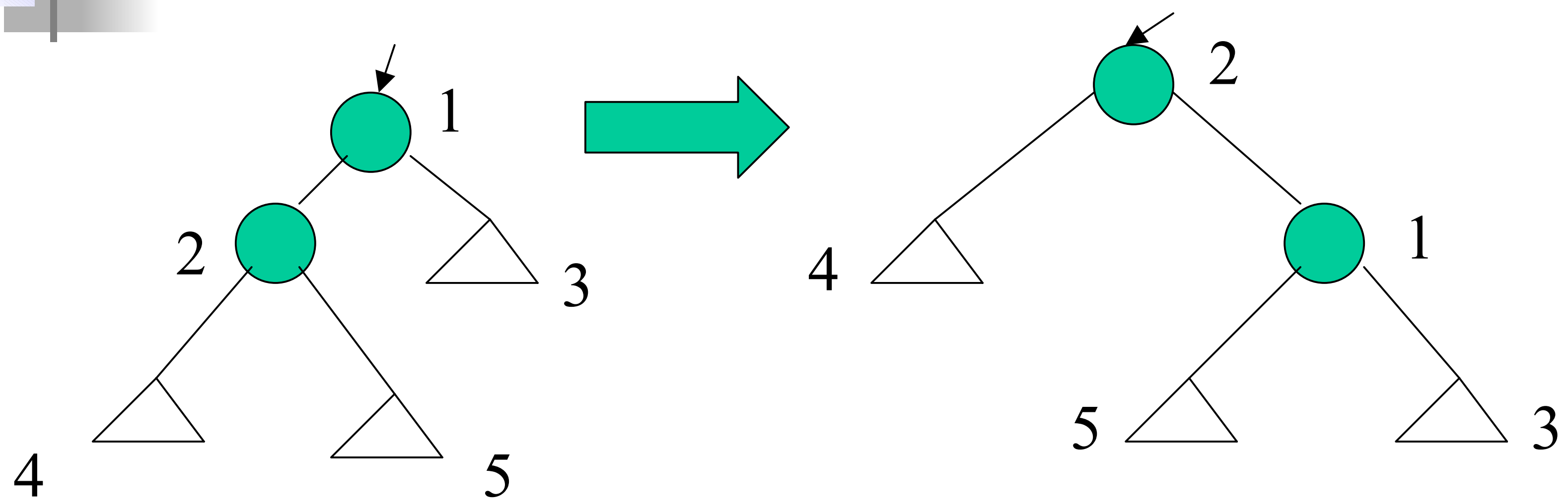
5. Delete a Key

Insert / Delete as in BST;
then rebalance the resultant tree
if necessary

AVL Tree Example



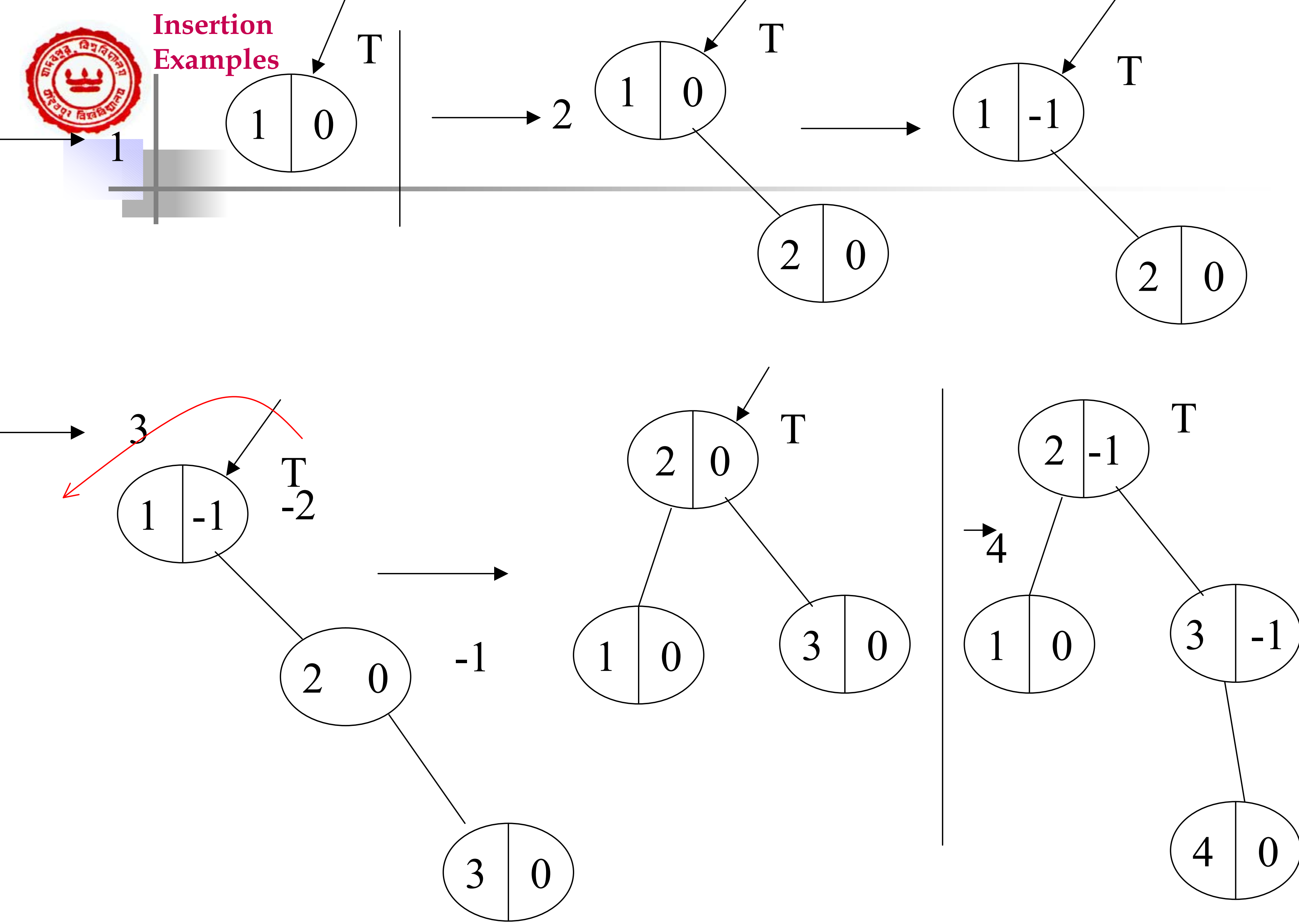
Rebalancing needs Rotation



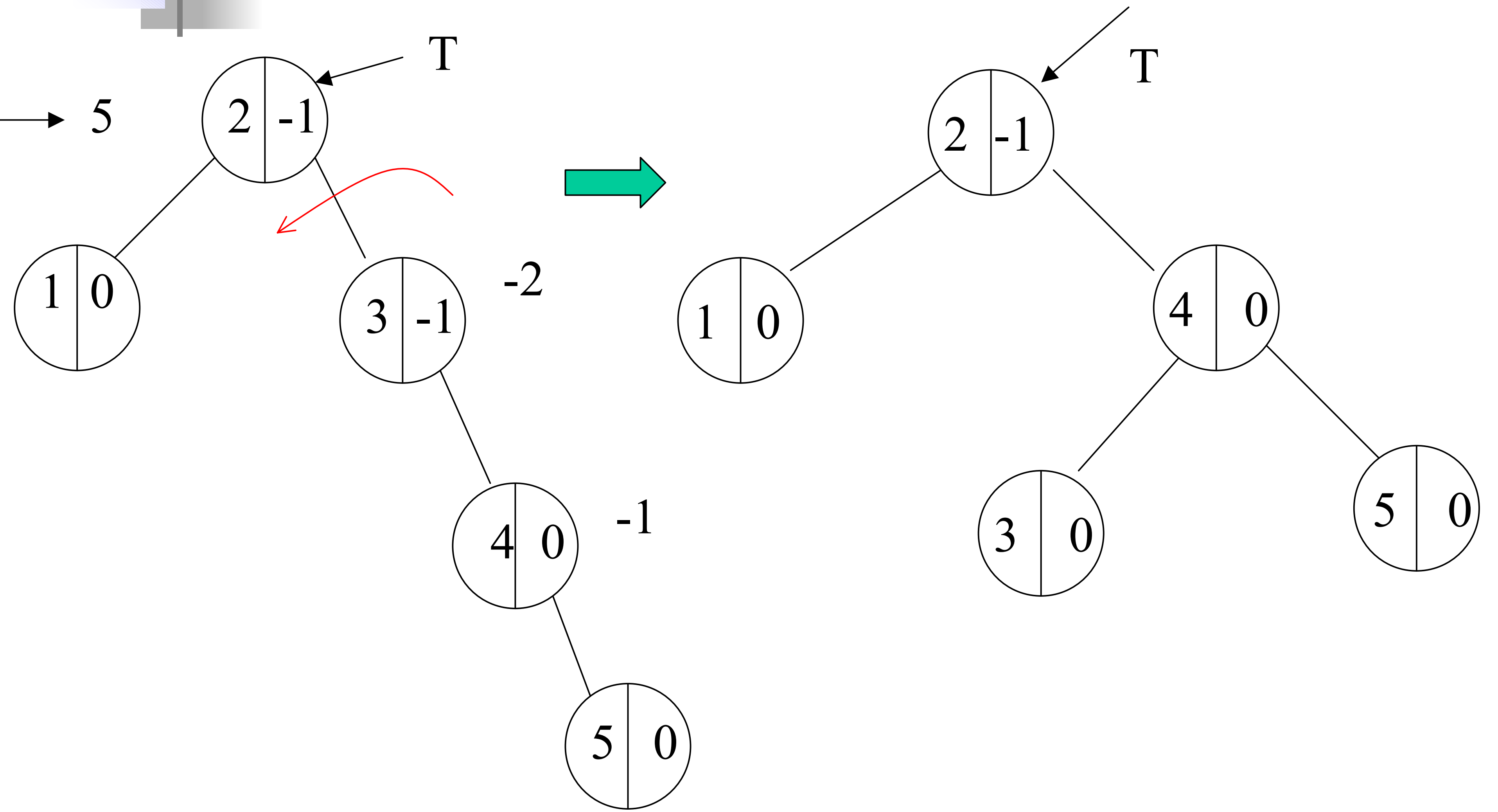
Right Rotation

```
avltree * rotate-right (avltree * t) {
        avltree * temp;
        temp = t → left;
        t → left = temp → right;
        temp → right = t;
        return temp;
}
```

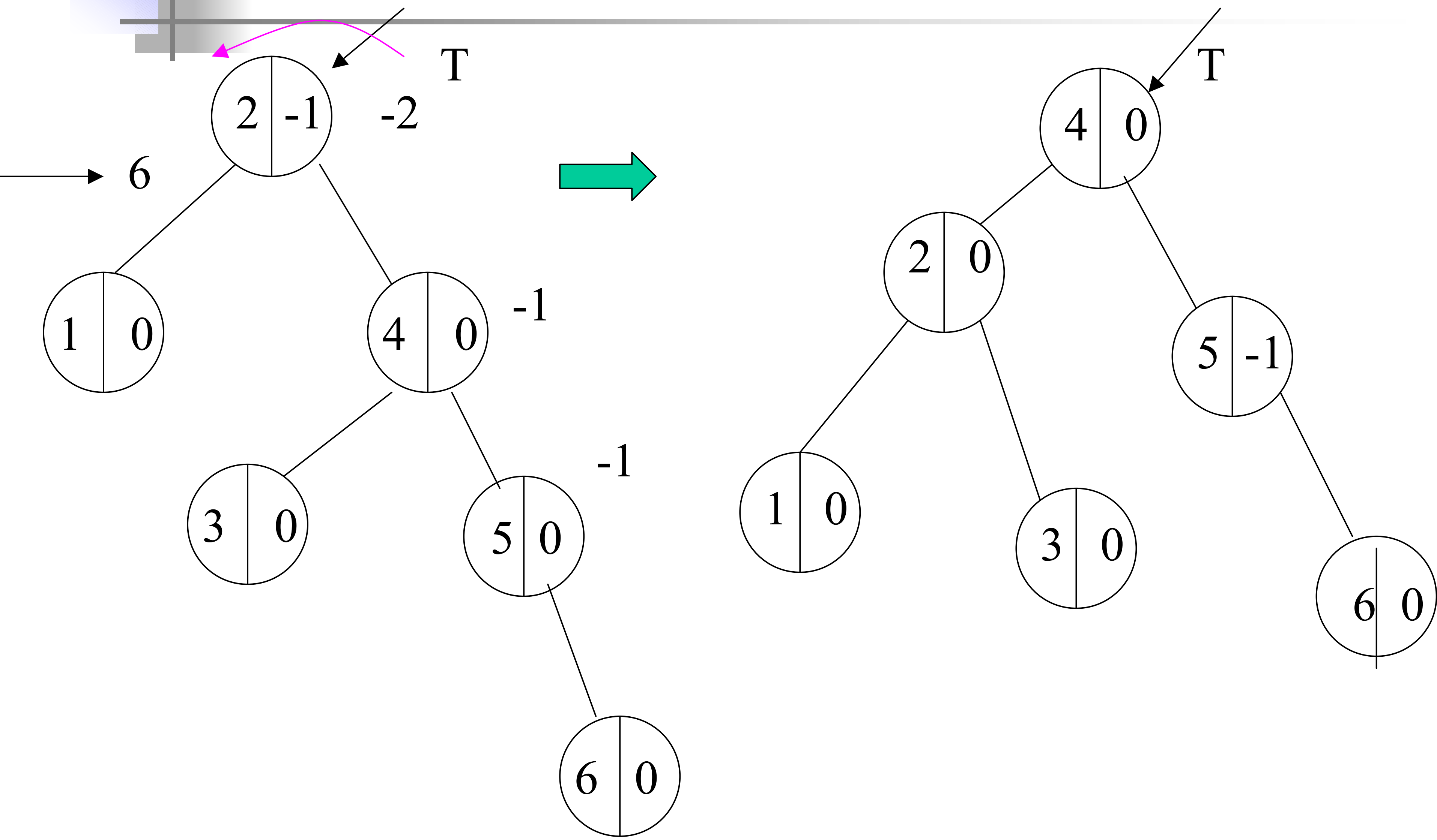
Insertion Examples



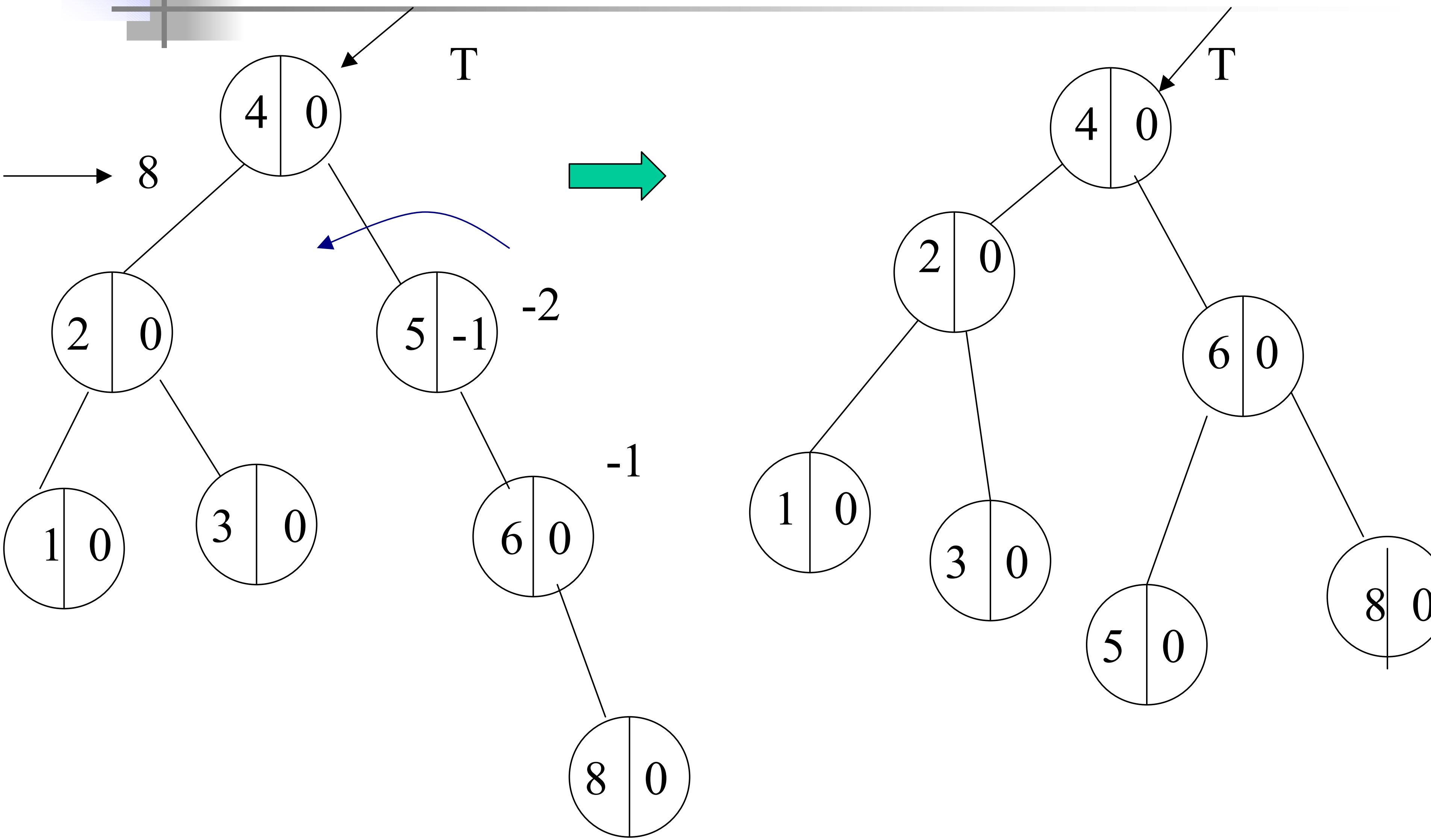
Insertion Examples …



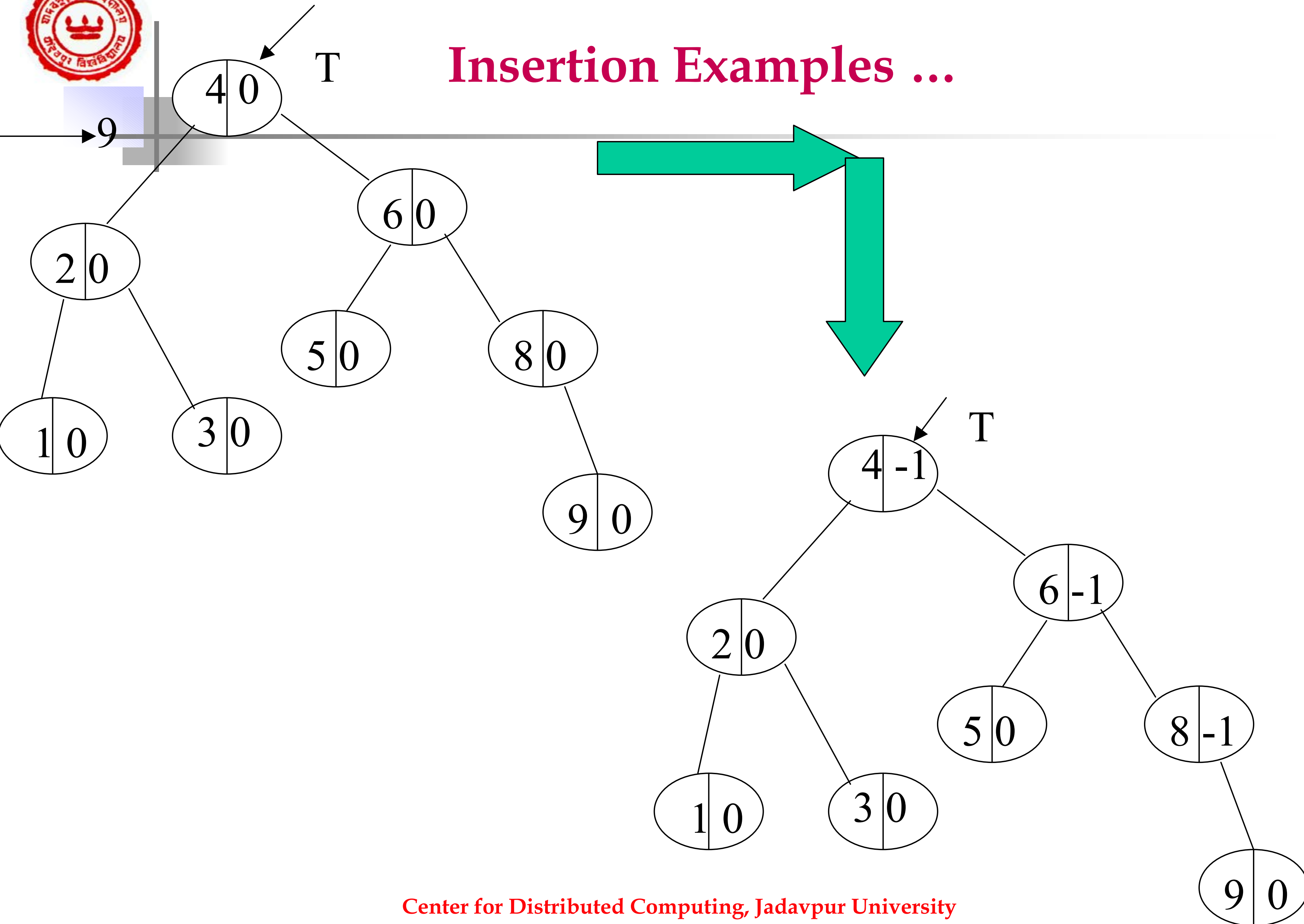
Insertion Examples …



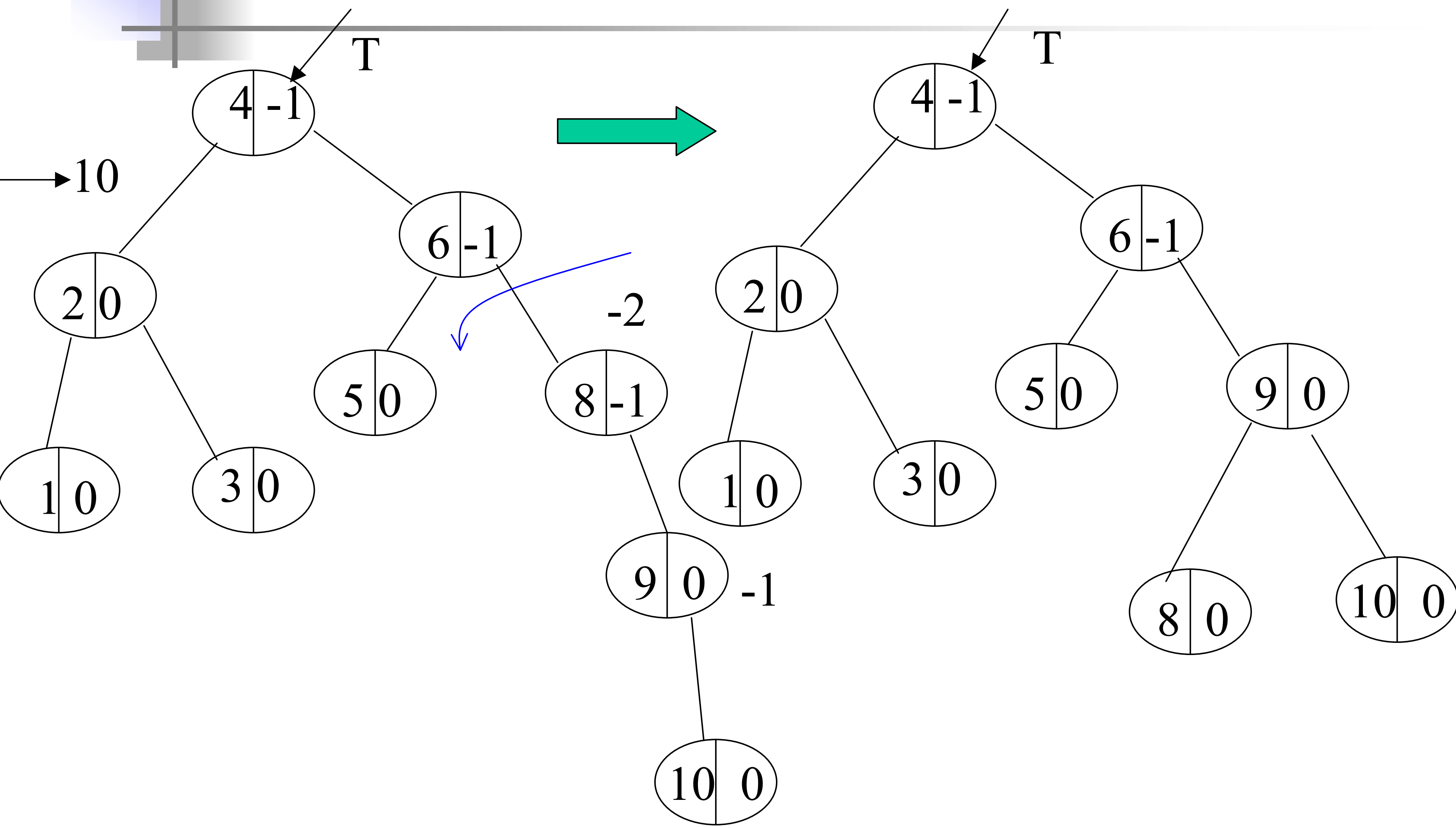
Insertion Examples …



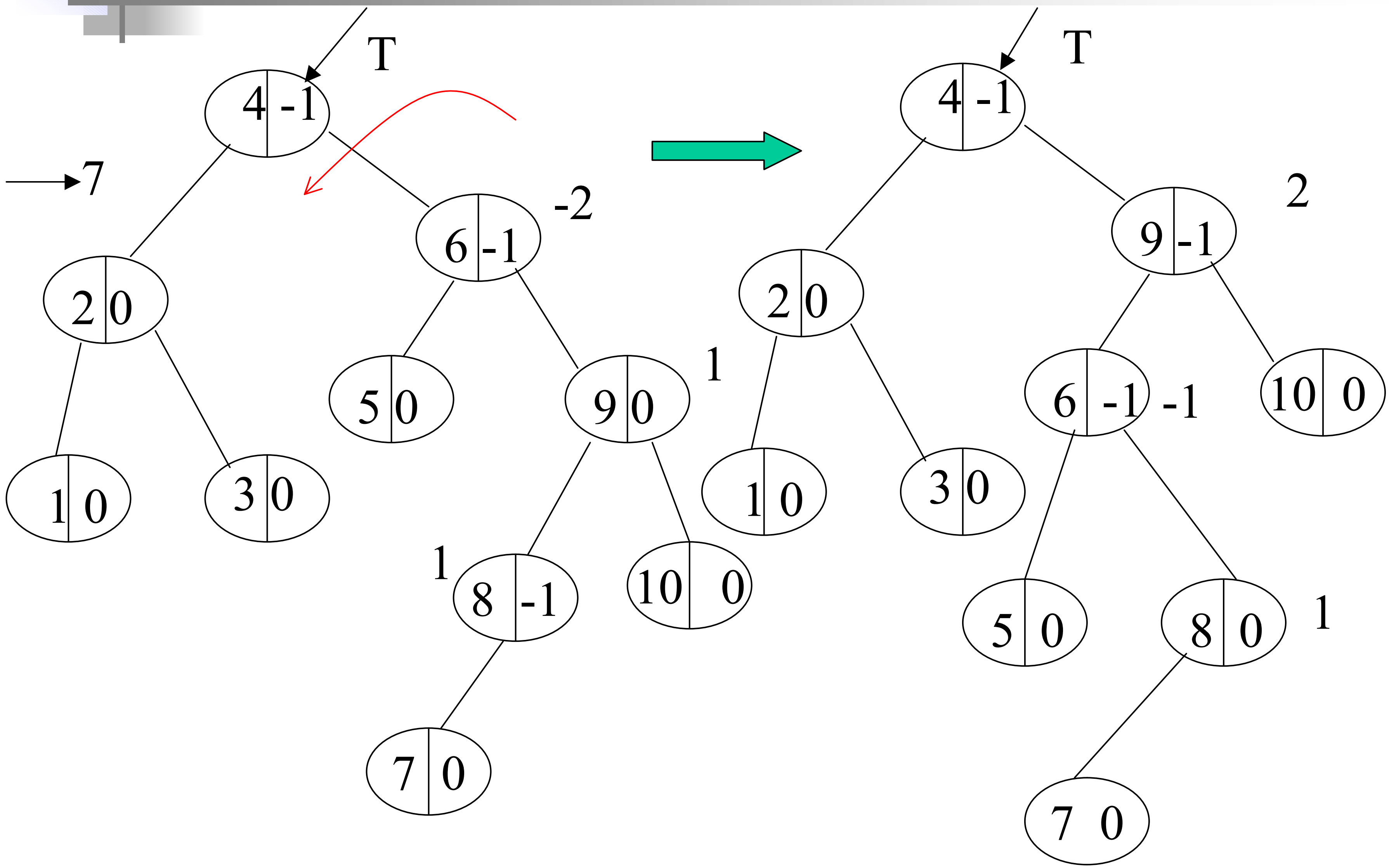
Insertion Examples …



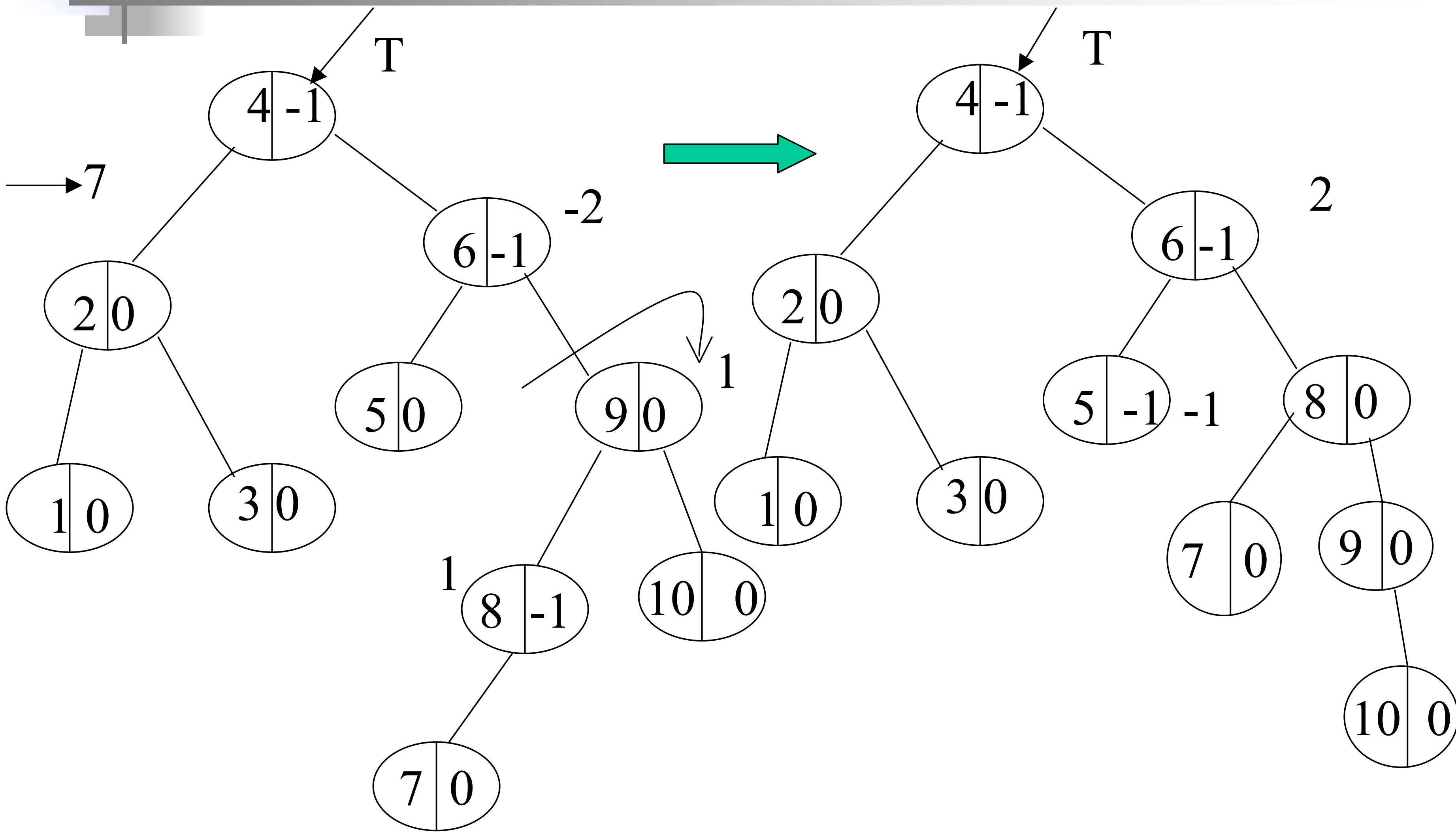
Insertion Examples …



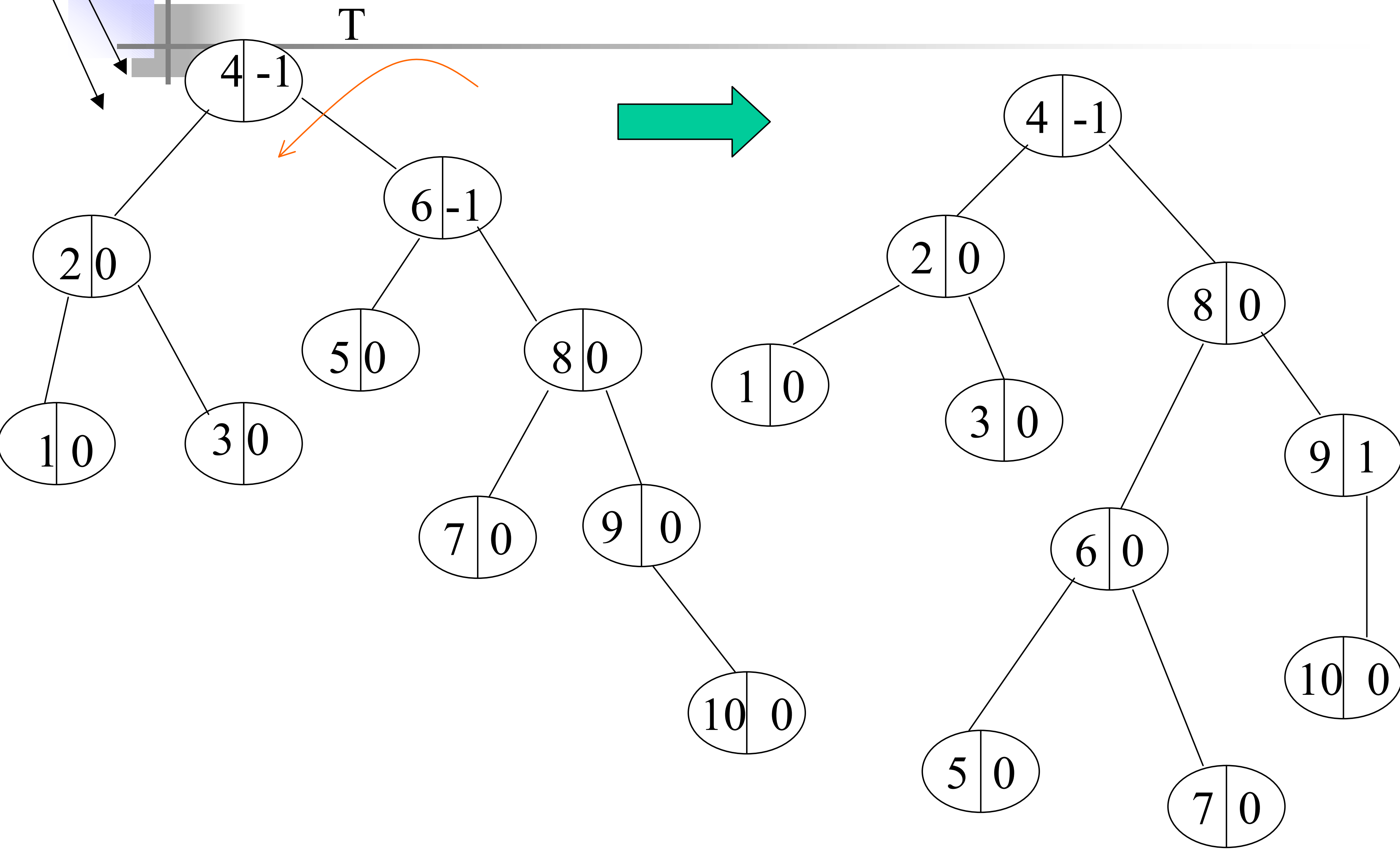
Tree remains unbalanced even after rotation



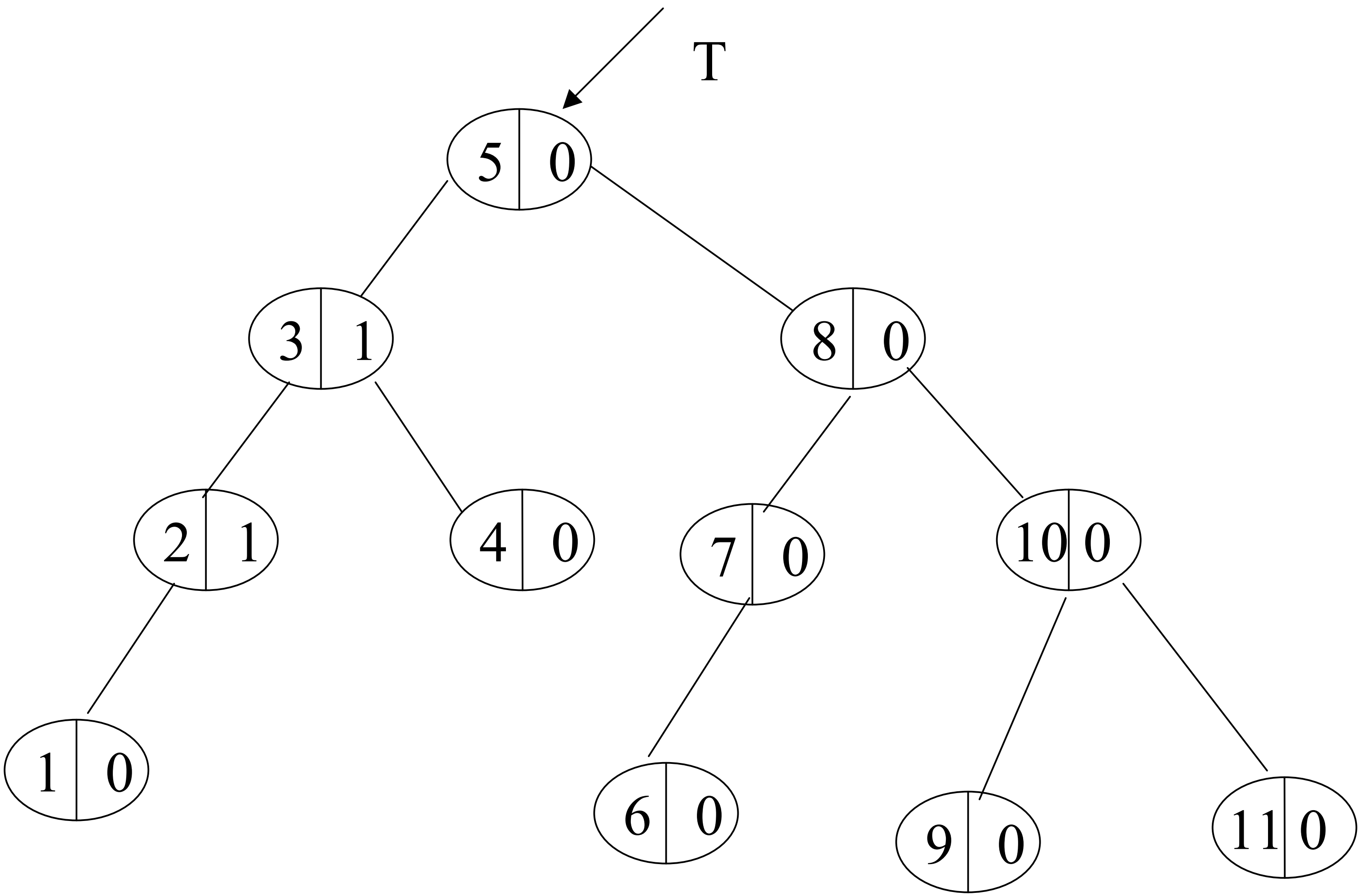
Right rotate the right child



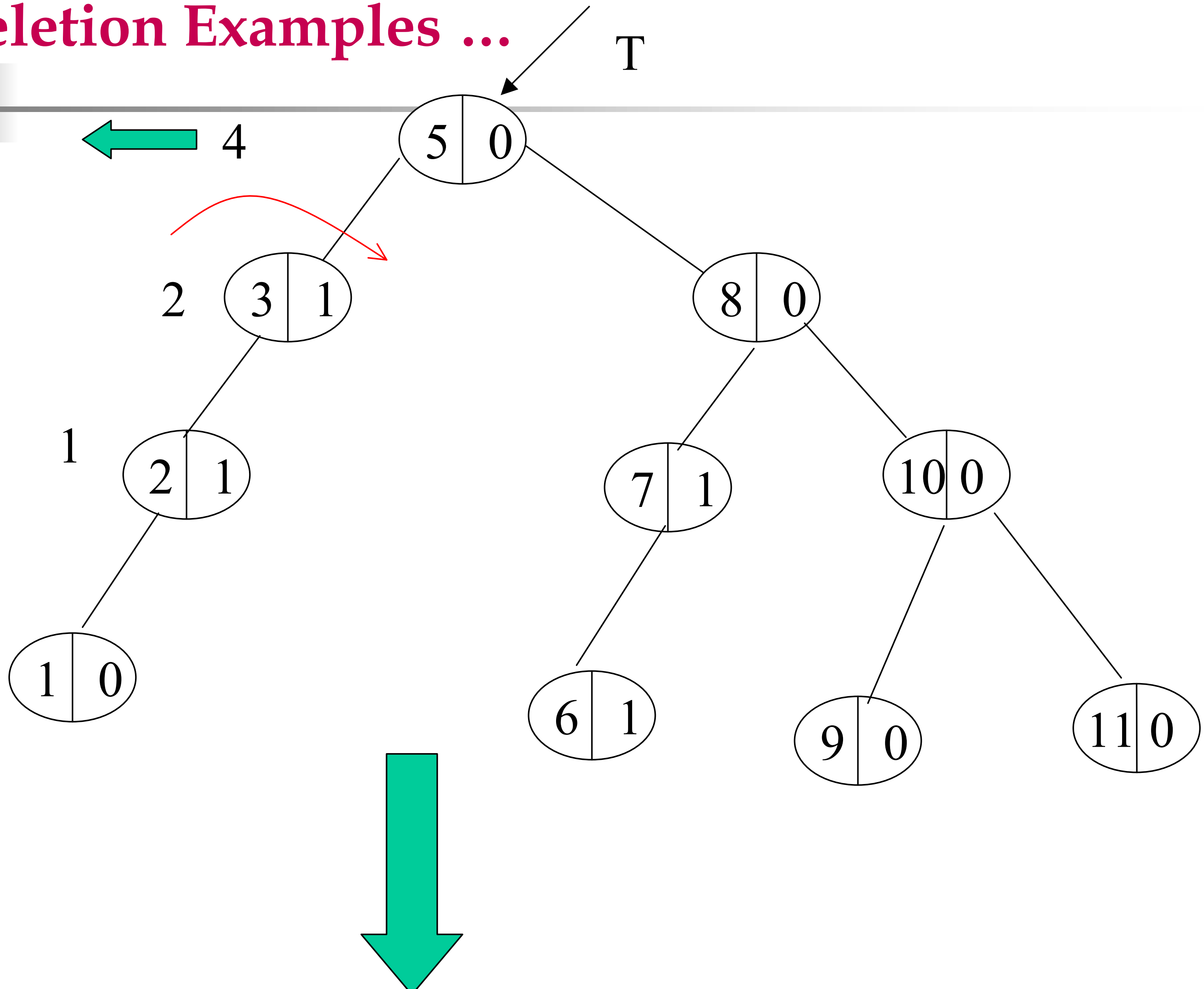
Double rotation: here, right rotation followed by left rotation



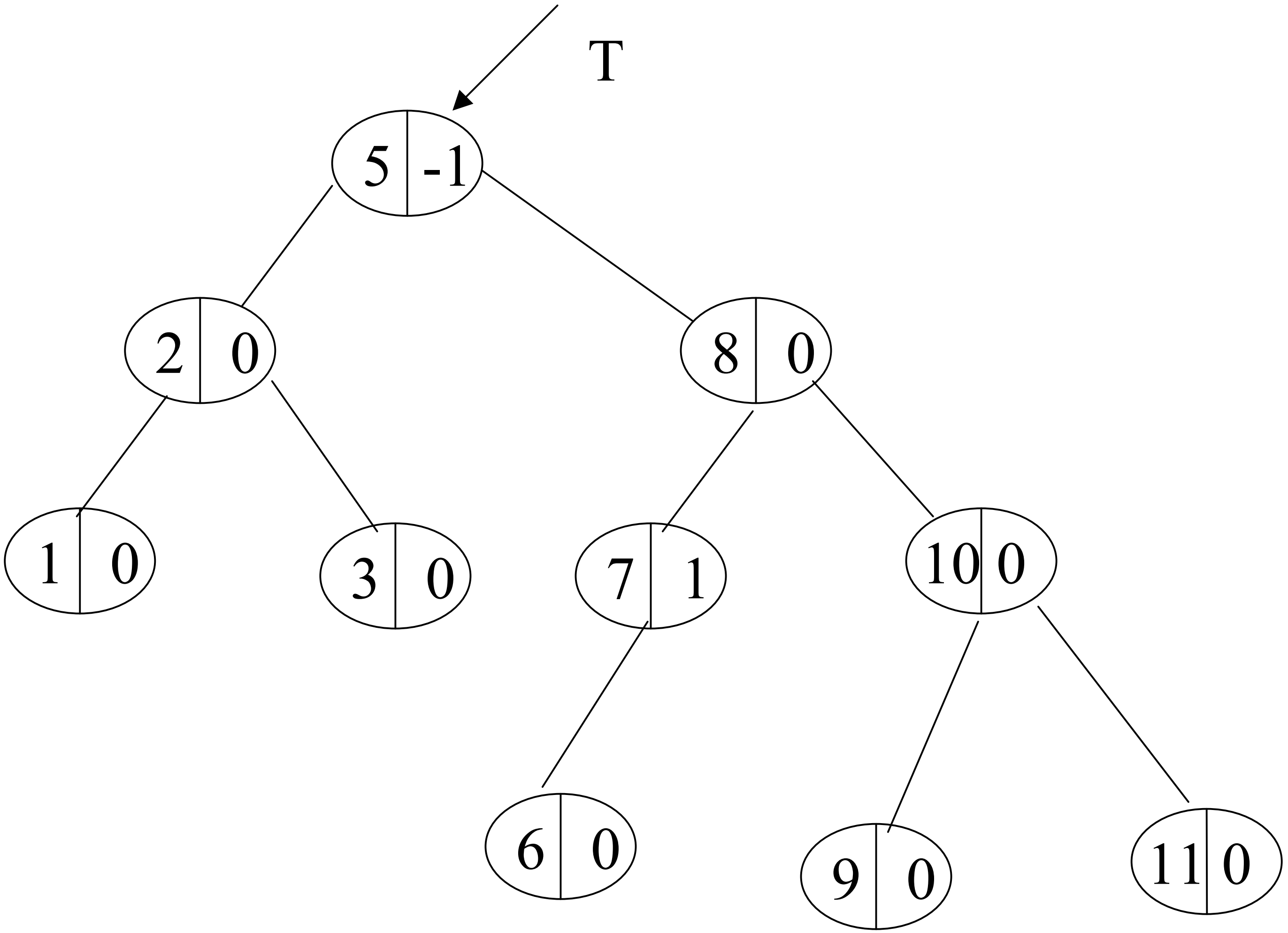
Deletion Examples



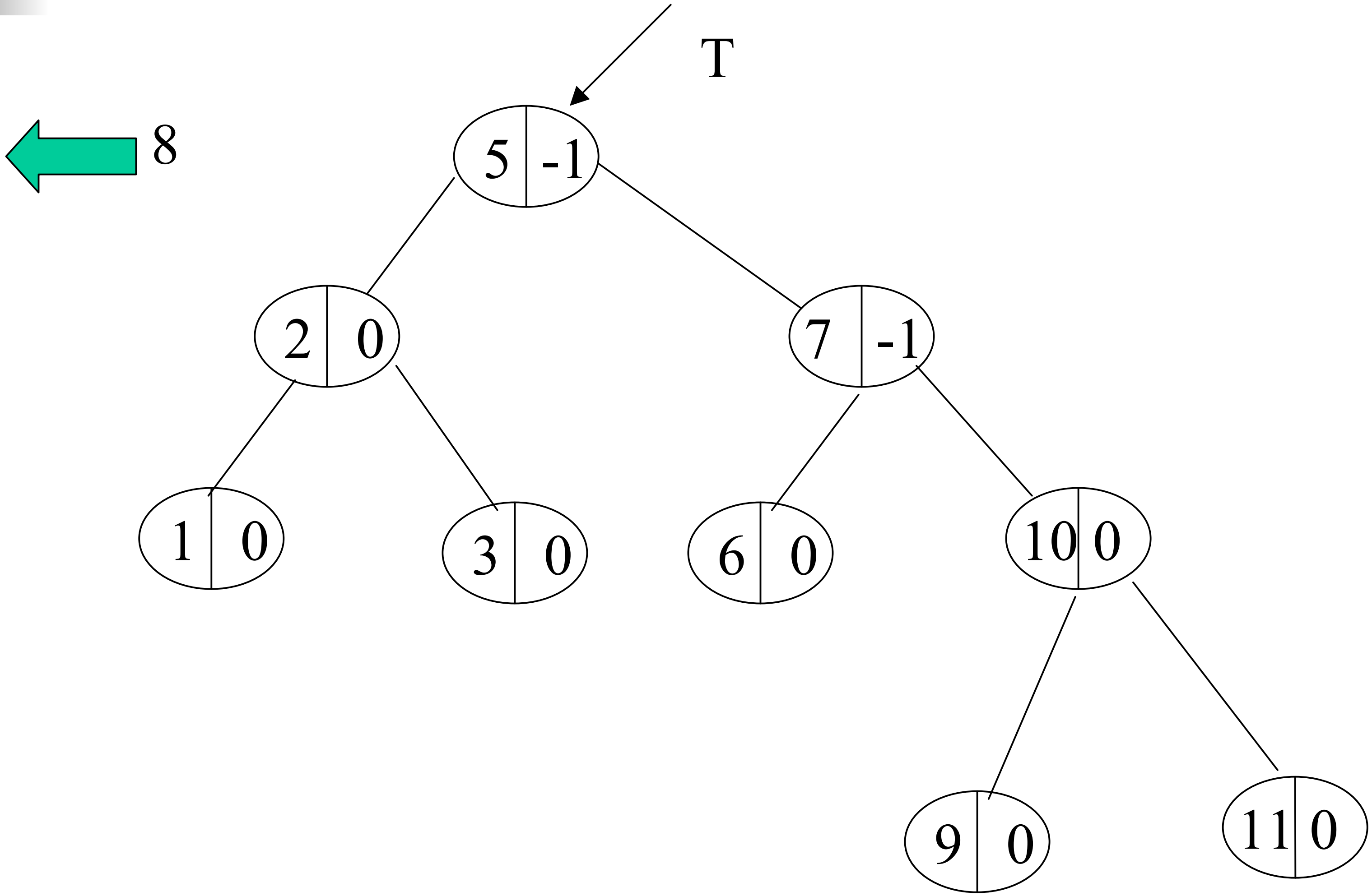
Deletion Examples …



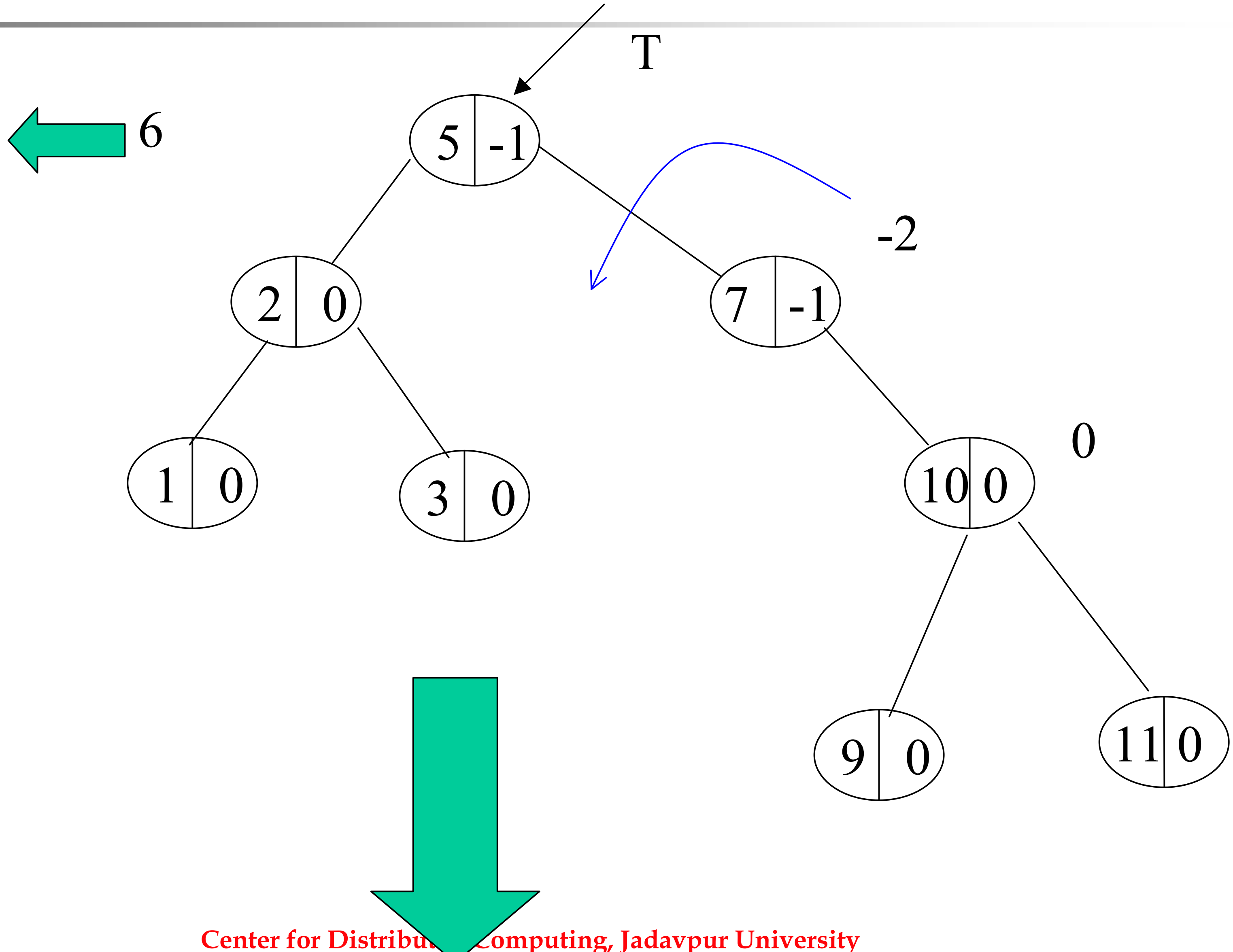
Deletion Examples …



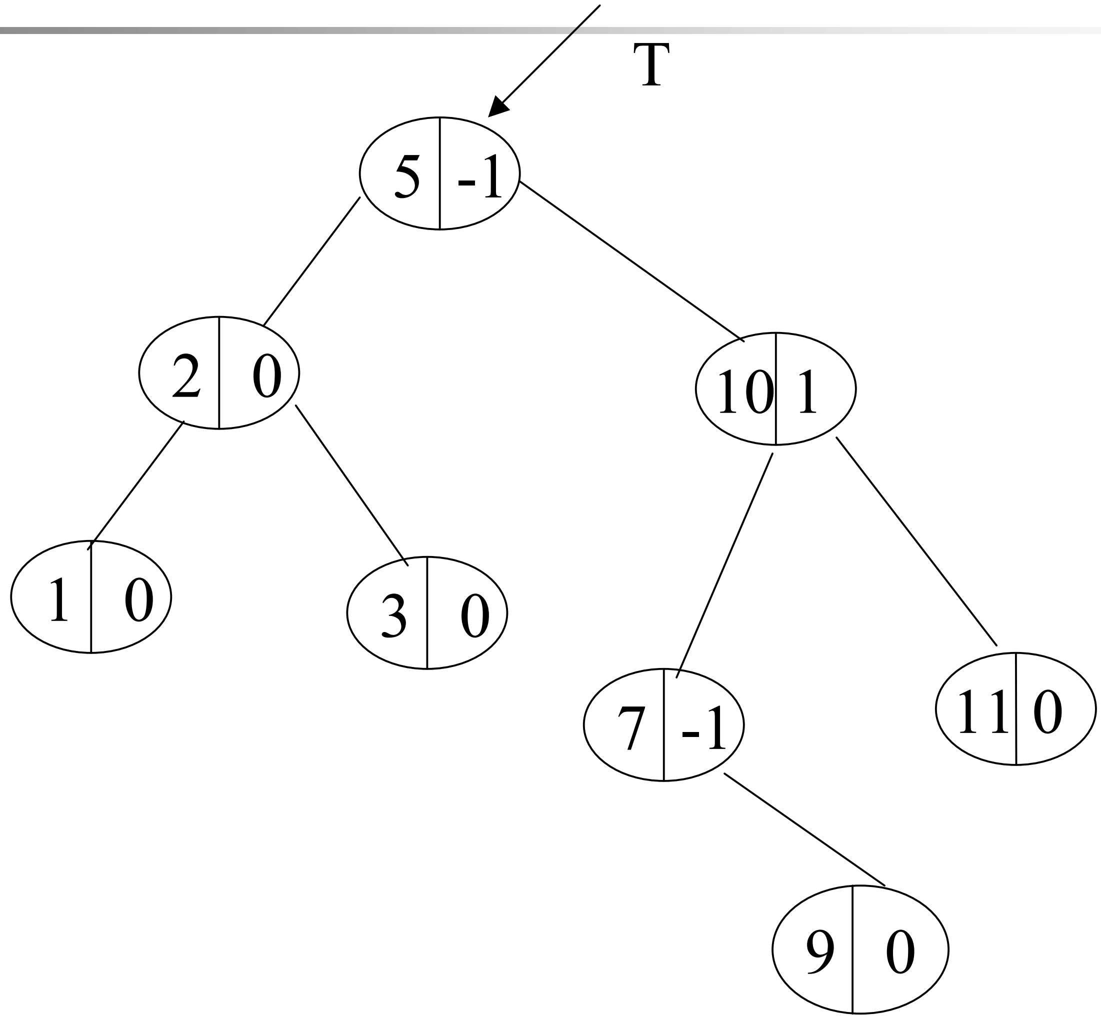
Deletion Examples …



Deletion Examples …



Deletion Examples …



Conclusion

- Height of a height-balanced (AVL) Tree is guaranteed to be O(log n), n being the no. of nodes.
- The insertion/deletion step takes at most O(log n) time.
- Each rebalancing step, i.e., rotation (possibly double rotation) and updation of BF takes a constant amount of time.
- The rebalancing may go up to the root. Thus, there can be at most O(log n) rebalancing steps.
- Thus the overall complexity of insertion/deletion is O(log n).