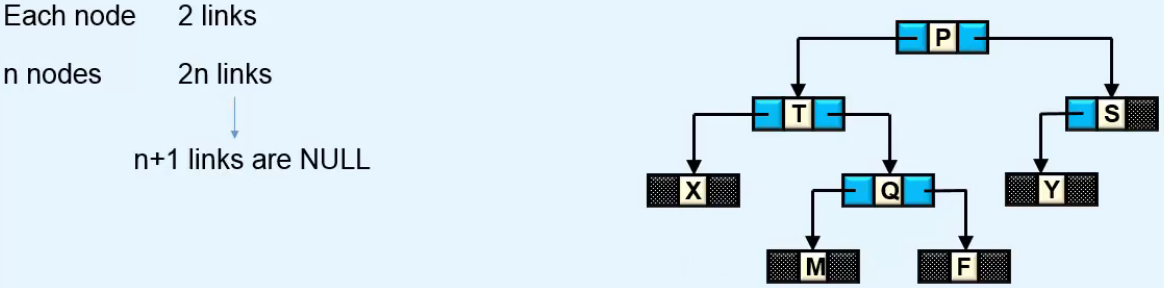
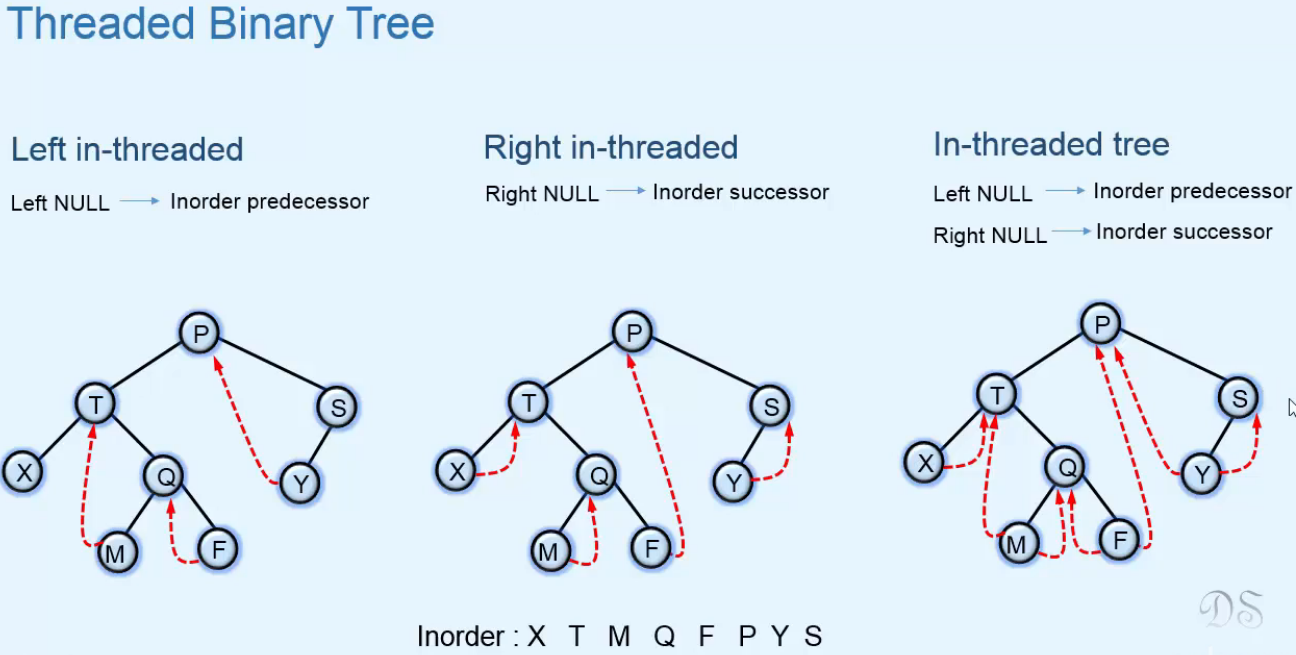
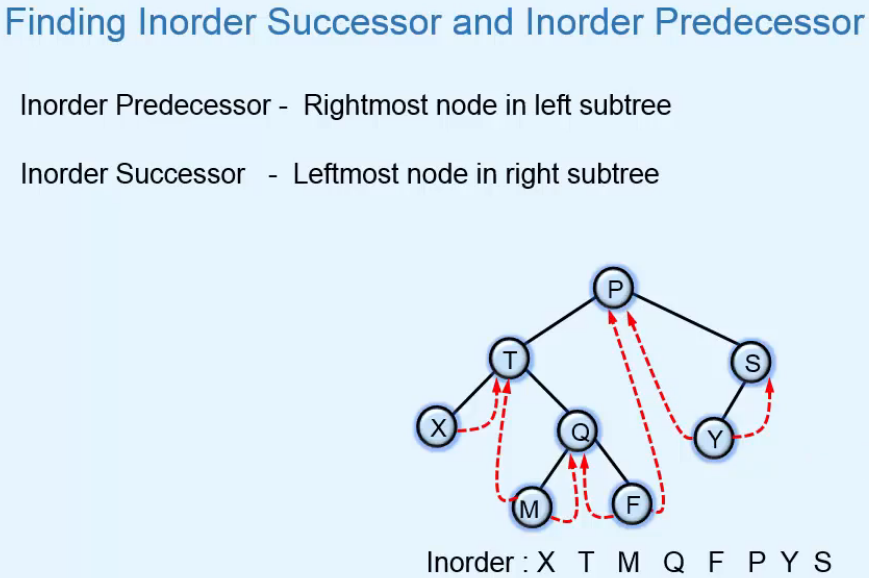
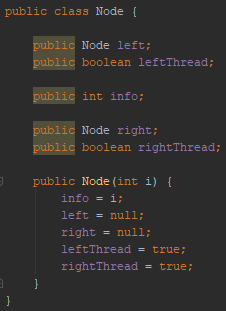
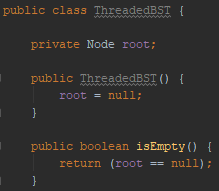
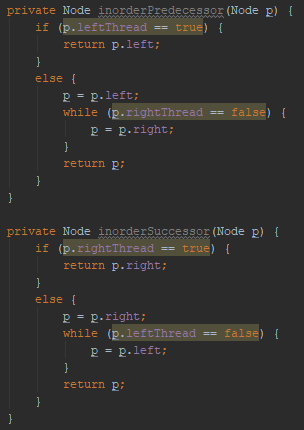
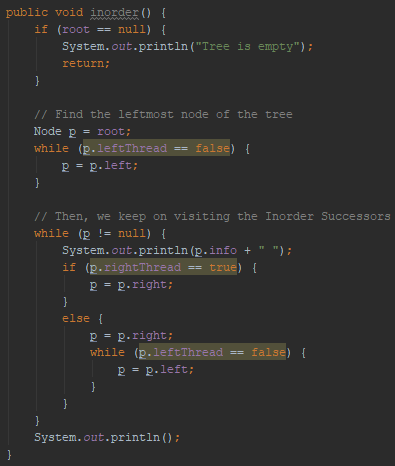
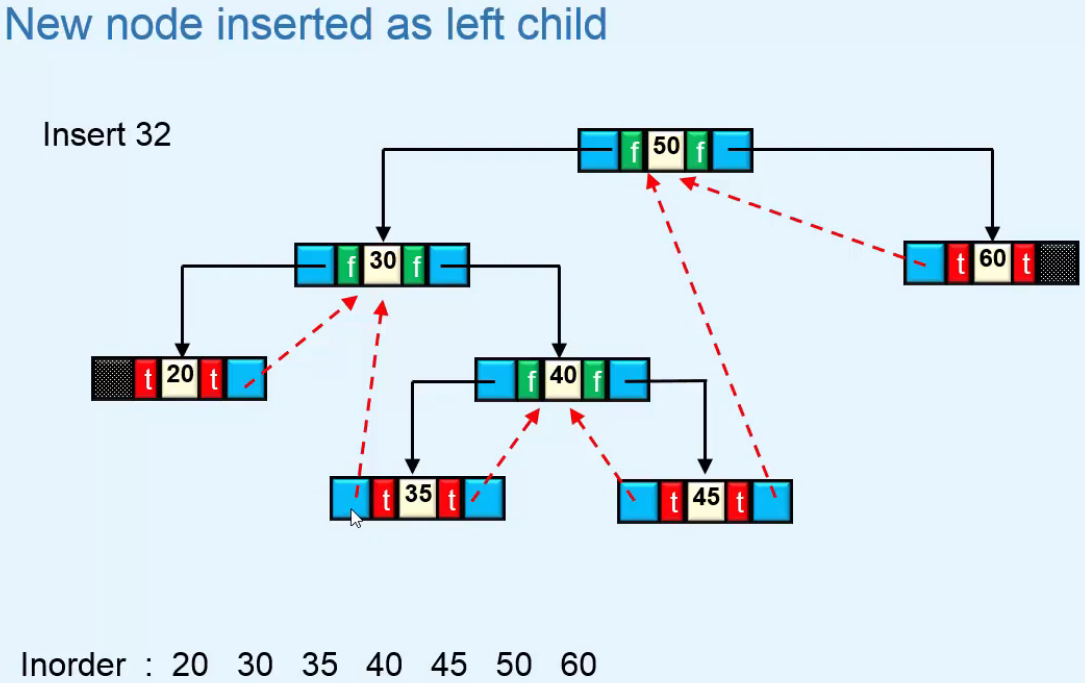
**Introduction**  
**Inorder Traversal of a Binary Tree**   
=> **Iterative** version requires **Explicit** **Stack**  
=> **Recursive** version uses a **Runtime Stack**  
\* **Traversal is generally a frequent operations and using a Stack is not efficient since it requires extra time and space**.  
**Inorder Traversal in Threaded Tree** => **No need for Stack**  
=> **Can be traversed more efficiently**.  
\* In a Binary Tree:  
=> each node => 2 links  
=> n nodes => 2n links  
=> n + 1 links are NULL  
  
=> Here we have 8 nodes and 9 NULL links.  
=> **Almost half the space that is allocated for links is wasted**.

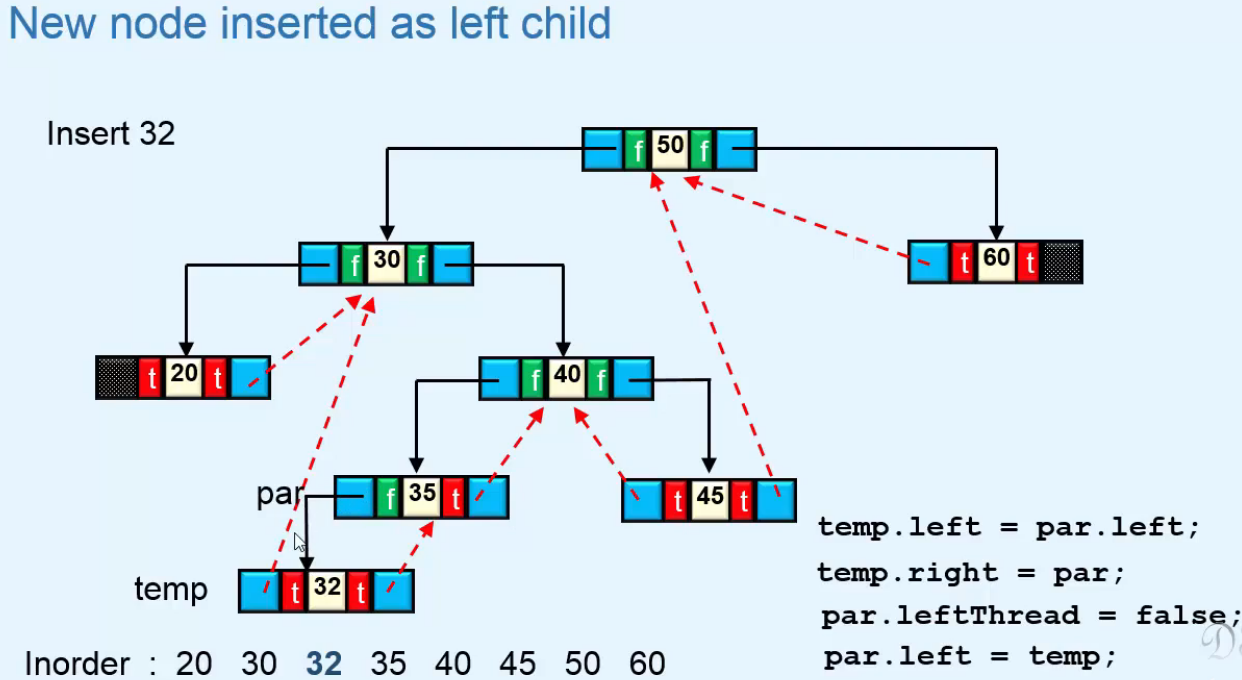
=> **We use this unused space to maintain threads**.

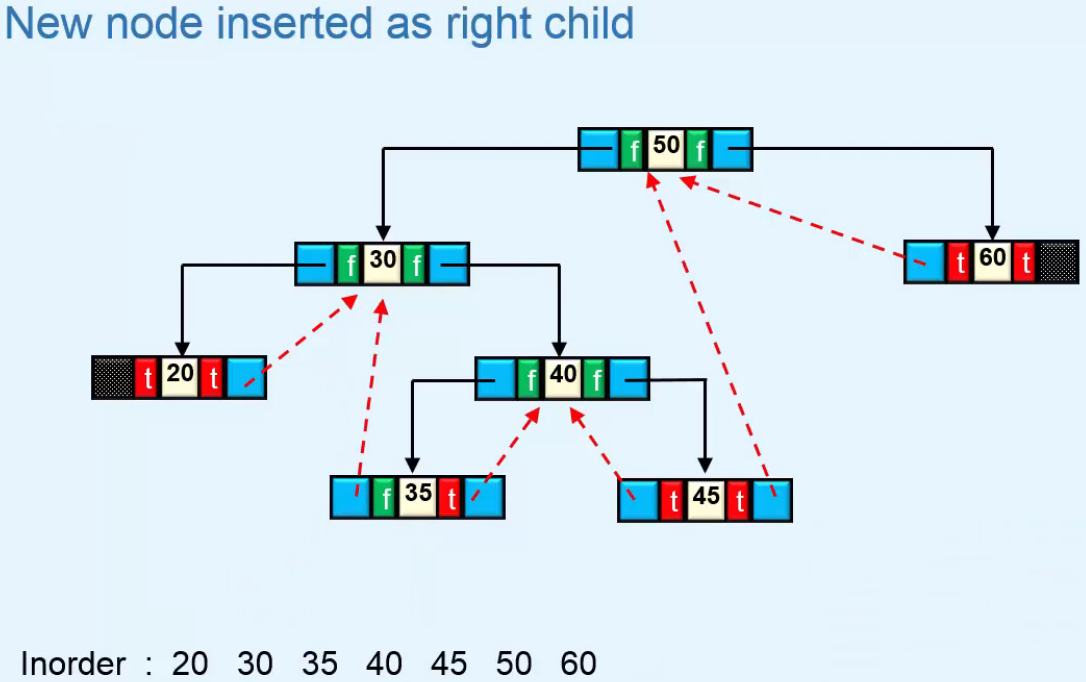
=> **Left null link refers to the inorder predecessor of the node**.  
=> **Right null link refers to the inorder successor of the node**.  
\* **The links that don’t refer to the tree nodes but refer to the predecesor or the successor, are called Threads and the Binary Tree that implements them is called a Threaded Binary Tree**.  
**Left in-threaded tree Right in-threaded tree In-threaded tree**  
**Left null -> Inorder predecessor** **right null -> Inorder successor** **Left null -> Inorder predecessor  
 right null -> Inorder successor**  
  
\* If a link has no Inorder predecessor or successor, it remains null.  
\* Left link of left-most node is null.  
\* Right link of right-most node is null.

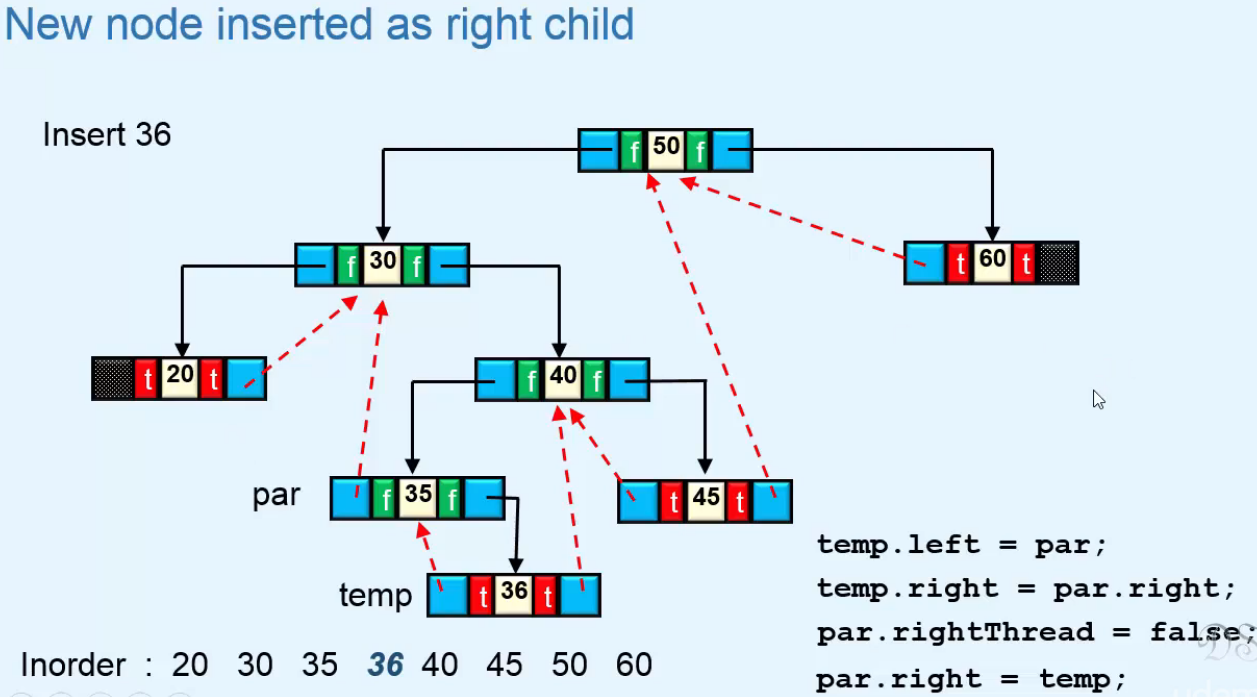
\* In our program we’ve taken a BST and used both left and right null links for threads.  
**In-threaded Binary Search Tree**  
\* We need some way to differentiate between normal child references and threads, so we use the boolean values leftThread, righThread.  
  
\* If the left link of a node is a thread, then there’s no need to do anything to find an Inorder Predecessor because in that case the left link refers to the Inorder Predecessor, we return left.  
\* If the right link of a node is a thread, then there’s no need to do anything to find an Inorder Successor because in that case the right link refers to the Inorder Successor, we return right.  
  
\* **For Inorder Traversal, we find the leftmost node of the tree**.  
\* **Then, we keep on visiting the Inorder Successors**.   
\* **The process stops when we get a node whose right child is null**.  
 

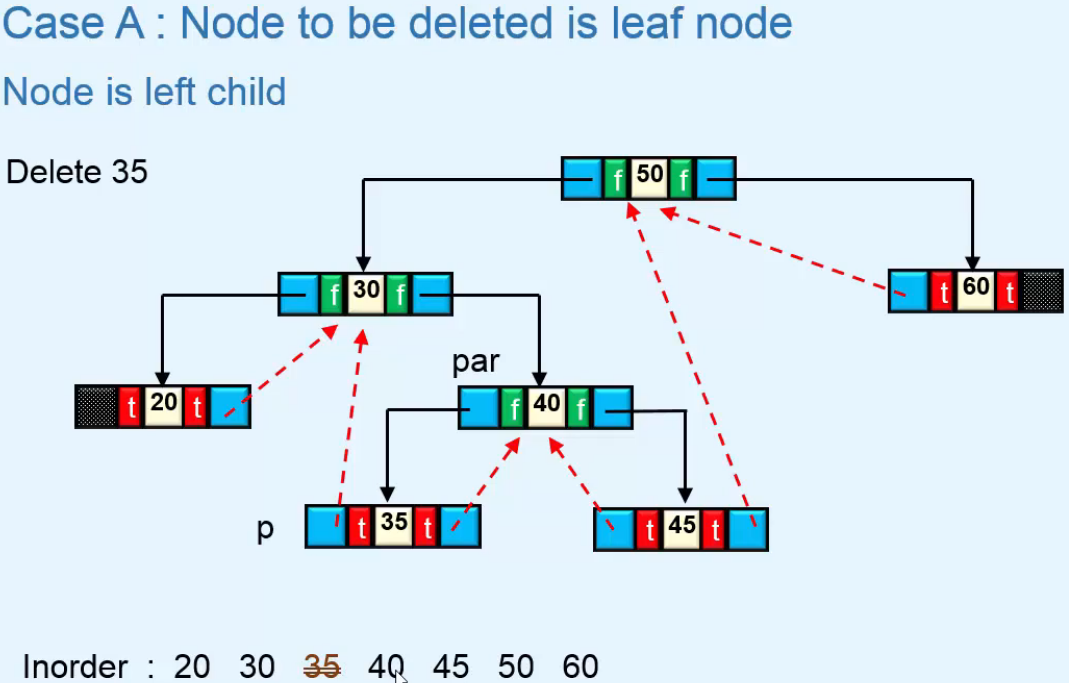
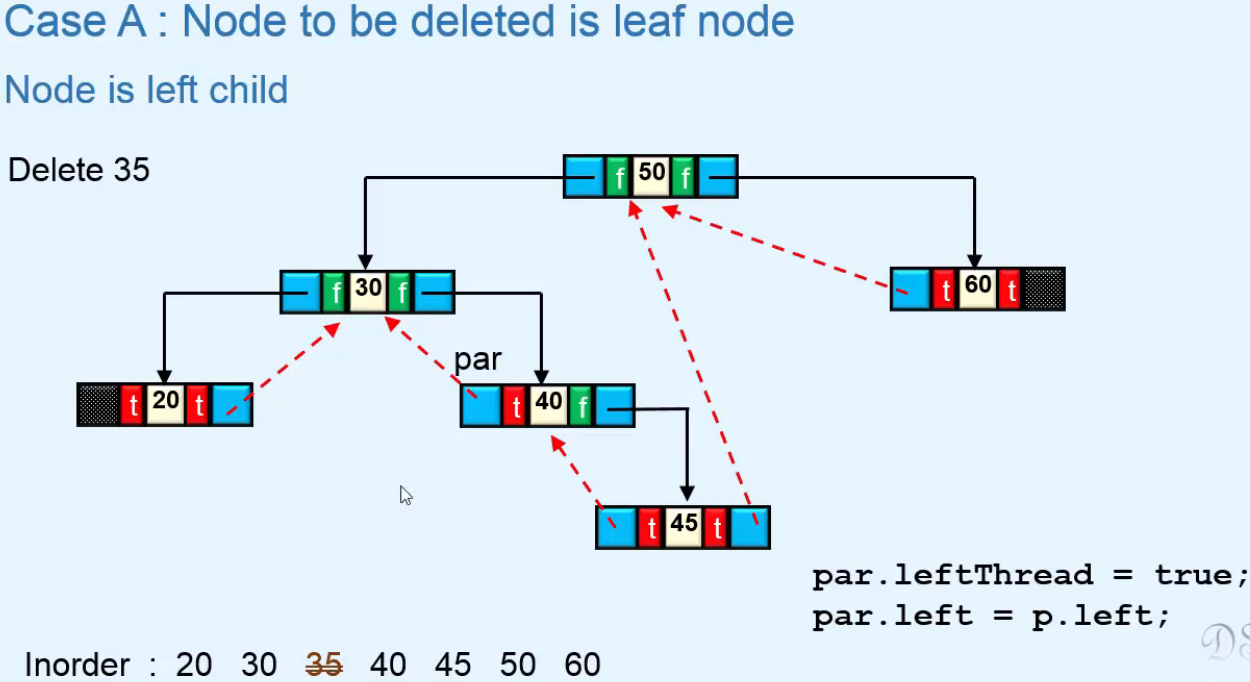
 

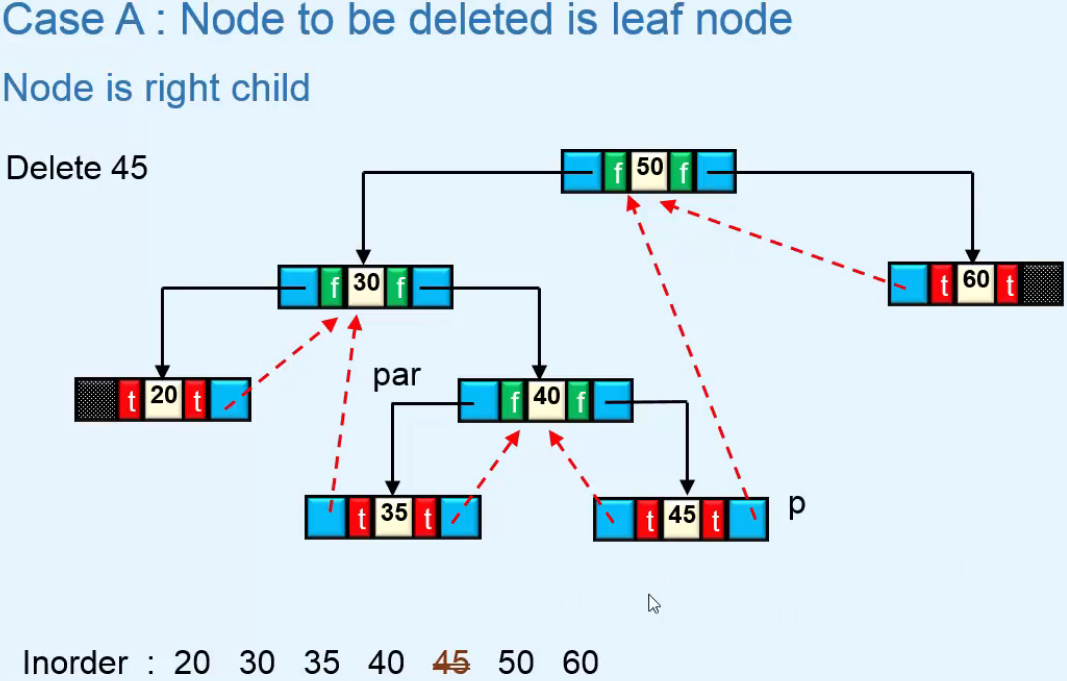
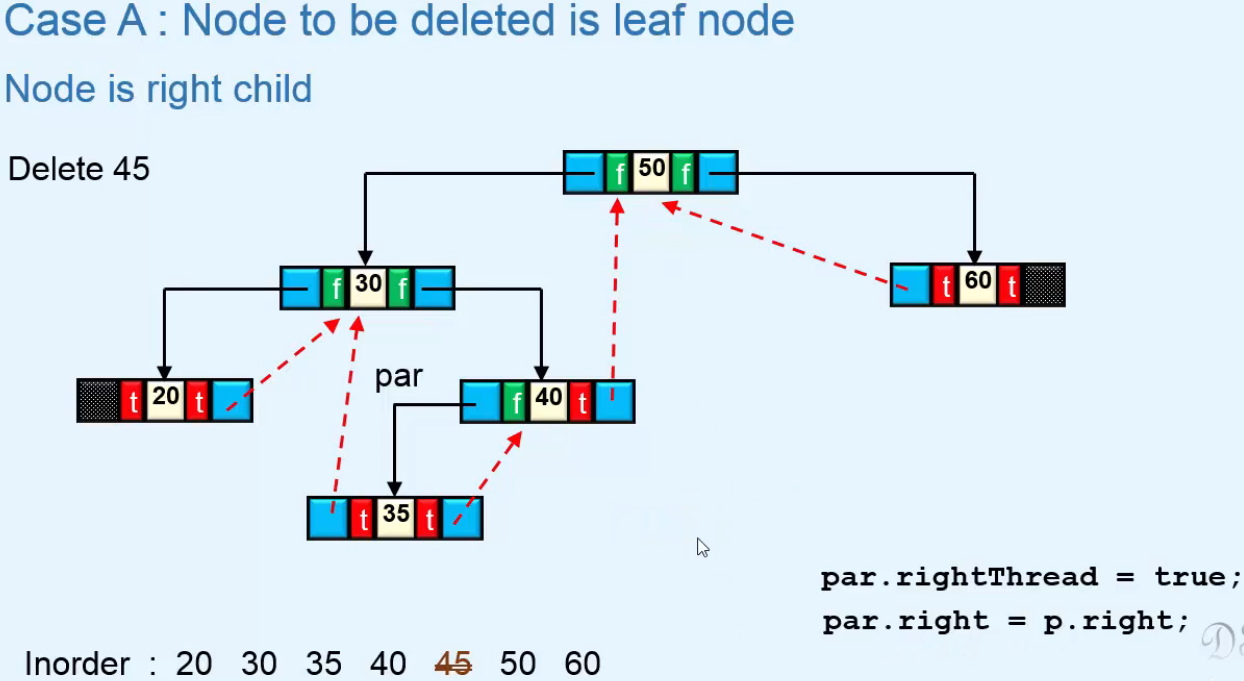
**Insertion**  
\* It’s done in the same way as in a BST, but here we have to do some extra work to adjust the threads after insertion.  
\* The WHILE loop stops when we’ve reached a thread or null child.  


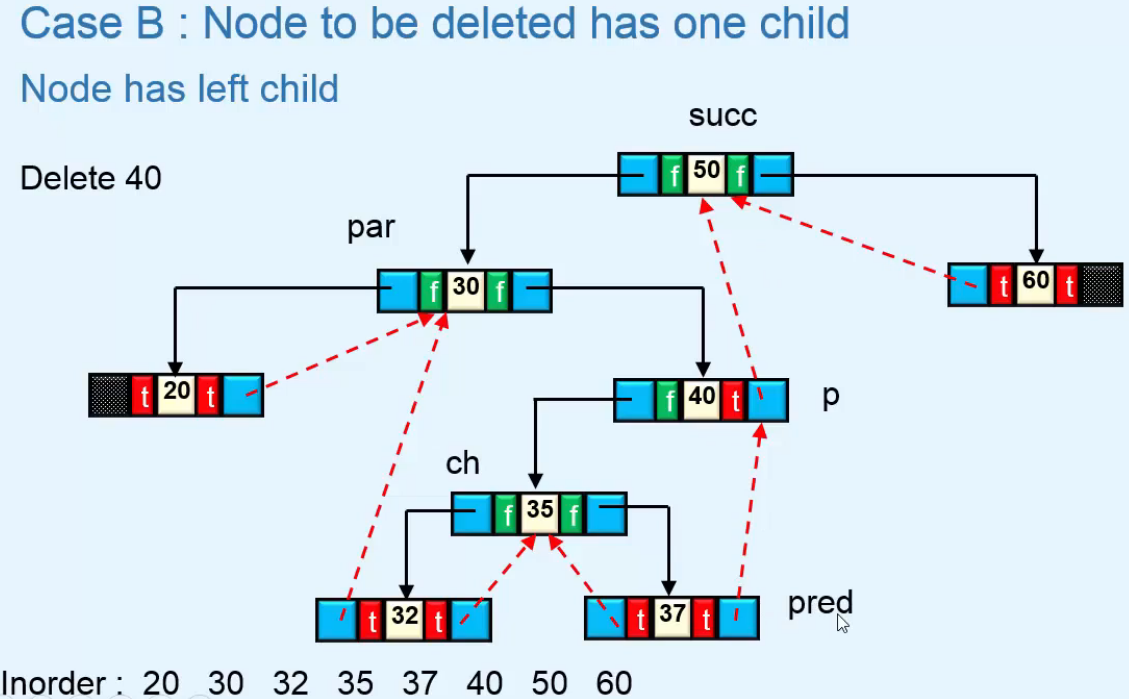


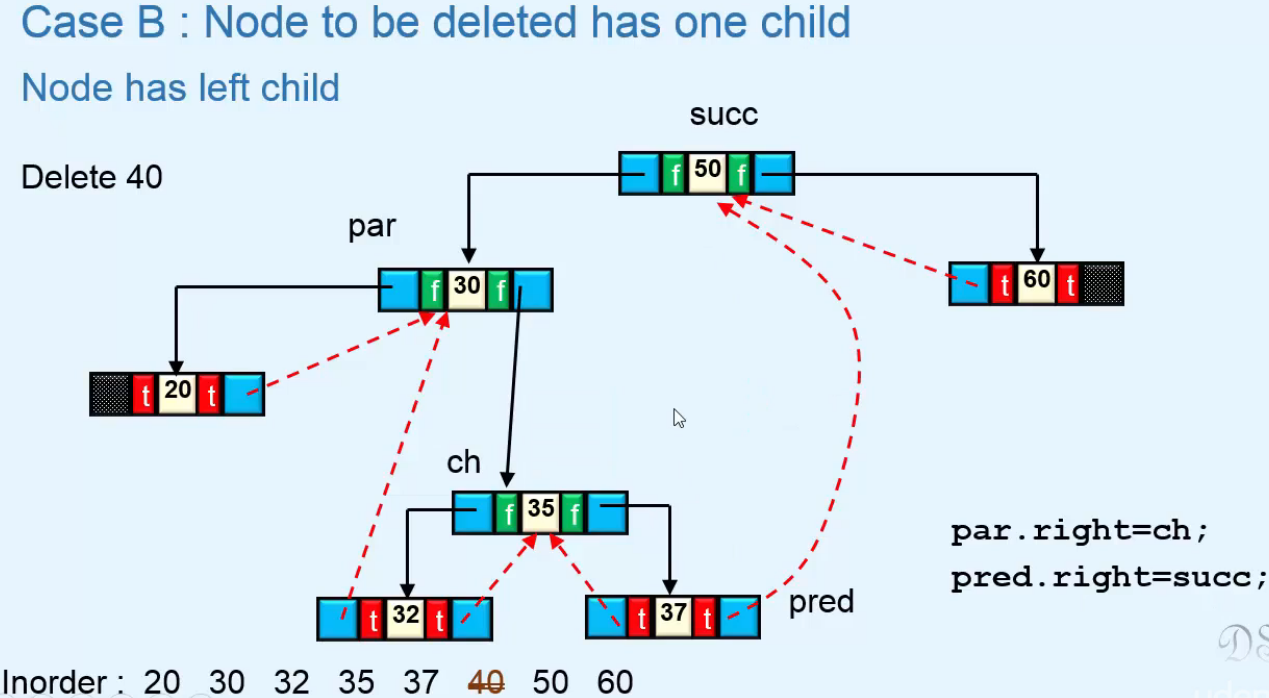


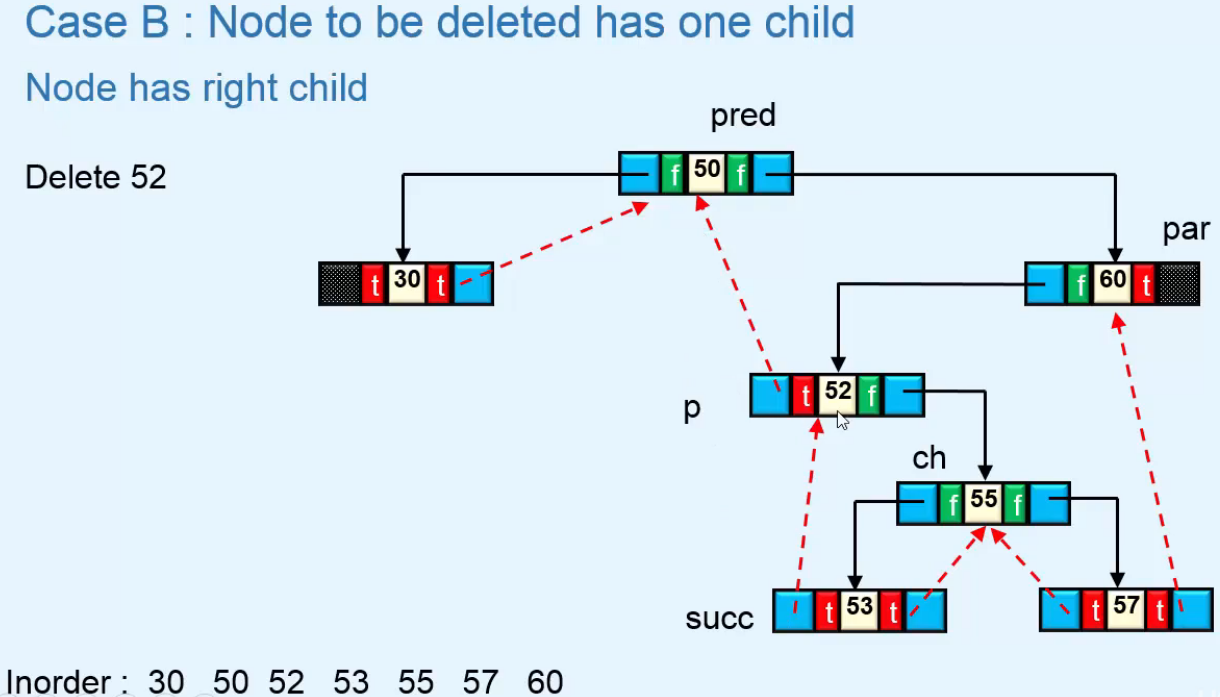


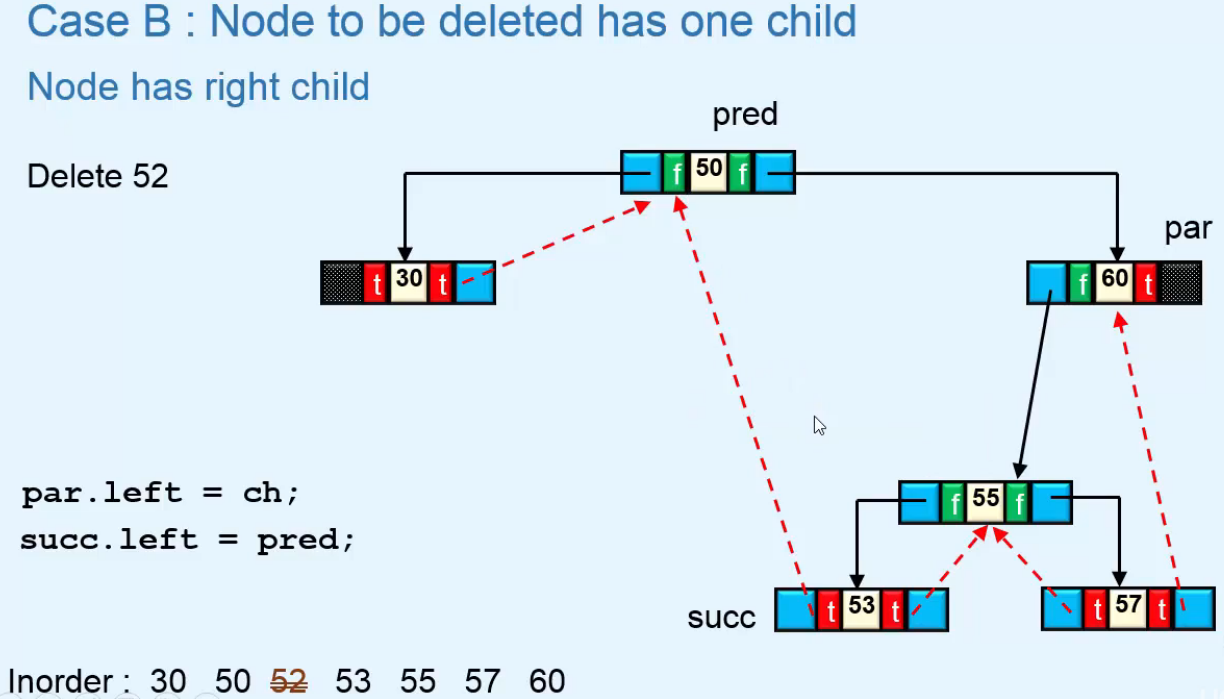
**Deletion**  
\* First we’re searching for the key that has to be deleted.  
\* So the WHILE loop part is the same as in Insertion.  
 









\* First we’re searching for the key that has to be deleted.  
\* So the WHILE loop part is the same as in Insertion.  
