Data Structures

IT 205

Dr. Manish Khare

Lecture – 21,22
08-Mar-2018

Threaded Binary Tree

Threaded Binary tree

- A binary tree is represented using array representation or linked list representation.
- When a binary tree is represented using linked list representation, if any node is not having a child we use NULL pointer in that position.
- In any binary tree linked list representation, there are more number of NULL pointer than actual pointers.
- Generally, in any binary tree linked list representation, if there are **2N** number of reference fields, then **N+1** number of reference fields are filled with NULL (**N+1 are NULL out of 2N**). This NULL pointer does not play any role except indicating there is no link (no child).

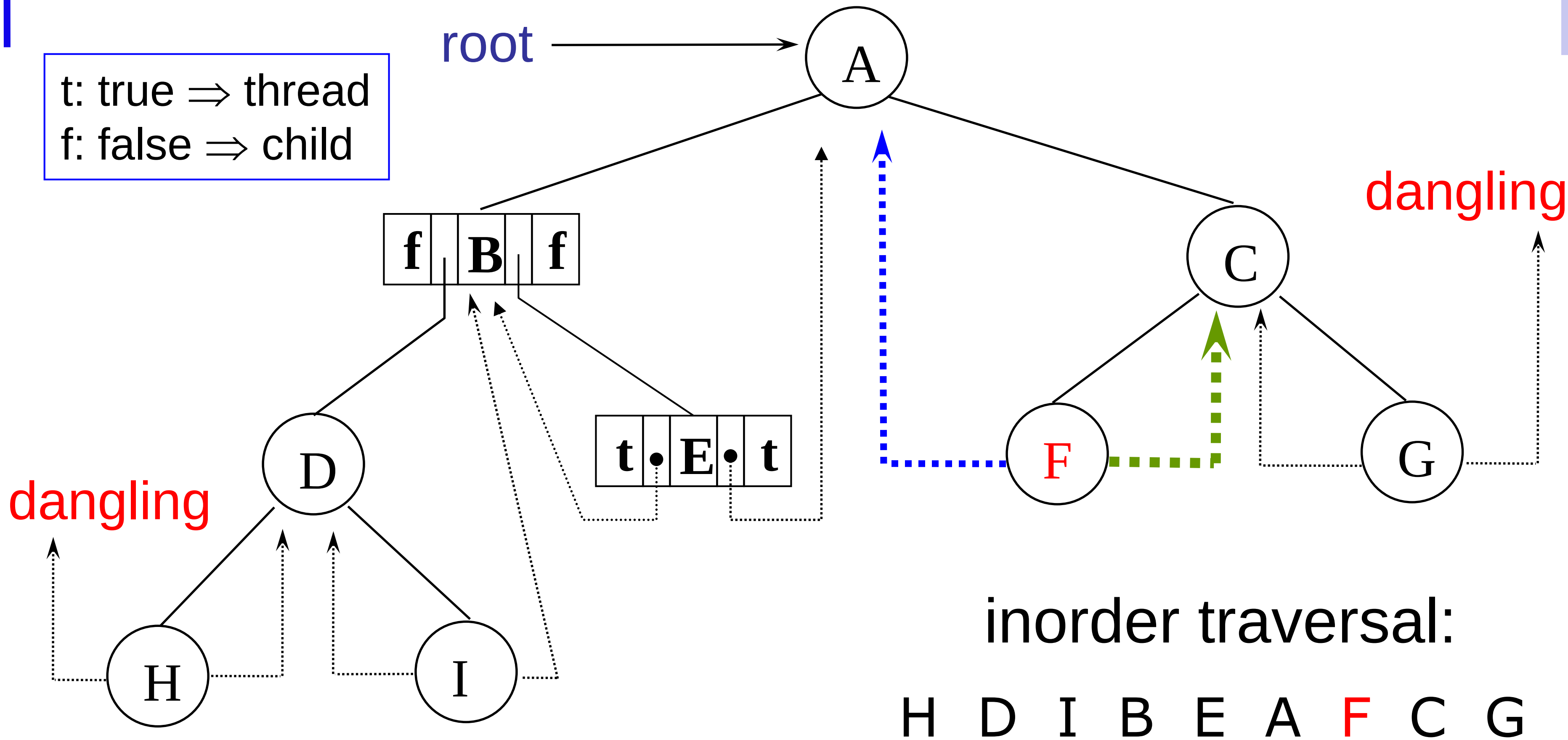
Threaded Binary tree

- Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor.

- If there is no in-order predecessor or in-order successor, then it point to root node.

Threaded Binary tree

- Rules for constructing the threads
  - If ptr->left_child is null,
    replace it with a pointer to the node that would be visited *before* `ptr` in an *inorder traversal*
  - If ptr->right_child is null,
    replace it with a pointer to the node that would be visited *after* `ptr` in an *inorder traversal*



inorder traversal:

H D I B E A F C G

- Two additional fields of the node structure, left-thread and right-thread
  - If ptr->left-thread=*TRUE*, then ptr->left-child contains a thread;
  - Otherwise it contains a pointer to the left child.
  - Similarly for the right-thread

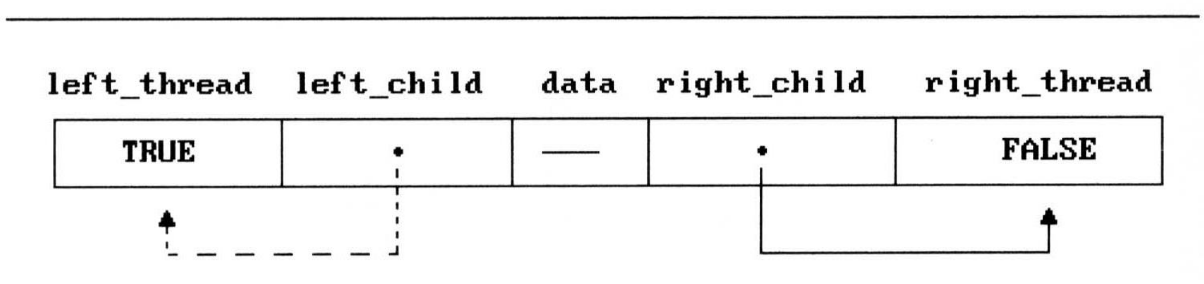


Figure 5.22: An empty threaded tree

- If we don't want the left pointer of H and the right pointer of G to be dangling pointers, we may create root node and assign them pointing to the root node

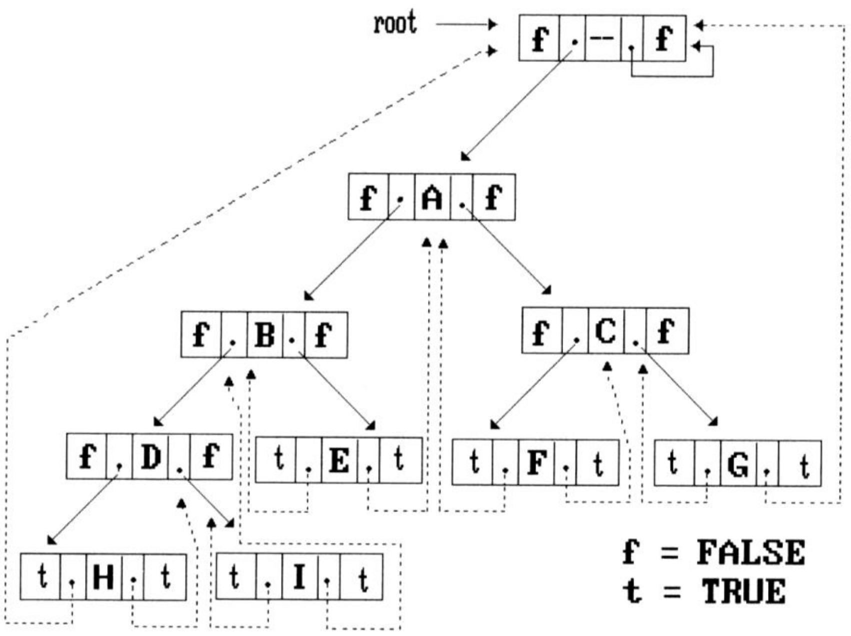


Figure 5.23: Memory representation of a threaded tree

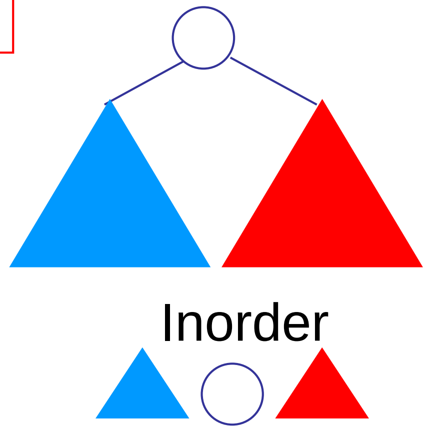
Inorder traversal of a threaded binary tree

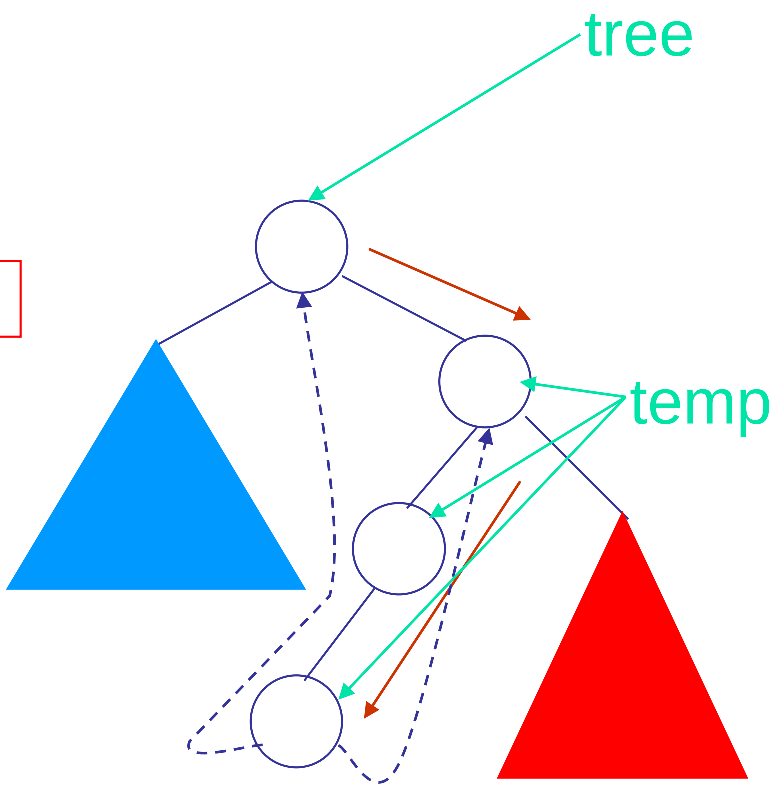
- By using of threads we can perform an inorder traversal without making use of a stack (simplifying the task)
- Now, we can follow the thread of any node, *ptr*, to the “next” node of inorder traversal

1. If *ptr->right_thread = TRUE*, the inorder successor of *ptr* is *ptr->right_child* by definition of the threads
2. Otherwise we obtain the inorder successor of *ptr* by following a path of left-child links from the right-child of *ptr* until we reach a node with *left_thread = TRUE*

Finding the inorder successor (next node) of a node

```
threaded_pointer insucc(threaded_pointer tree){
    threaded_pointer temp;
    temp = tree->right_child;
    if (!tree->right_thread)
      while (!temp->left_thread)
            temp = temp->left_child;
    return temp;
}
```



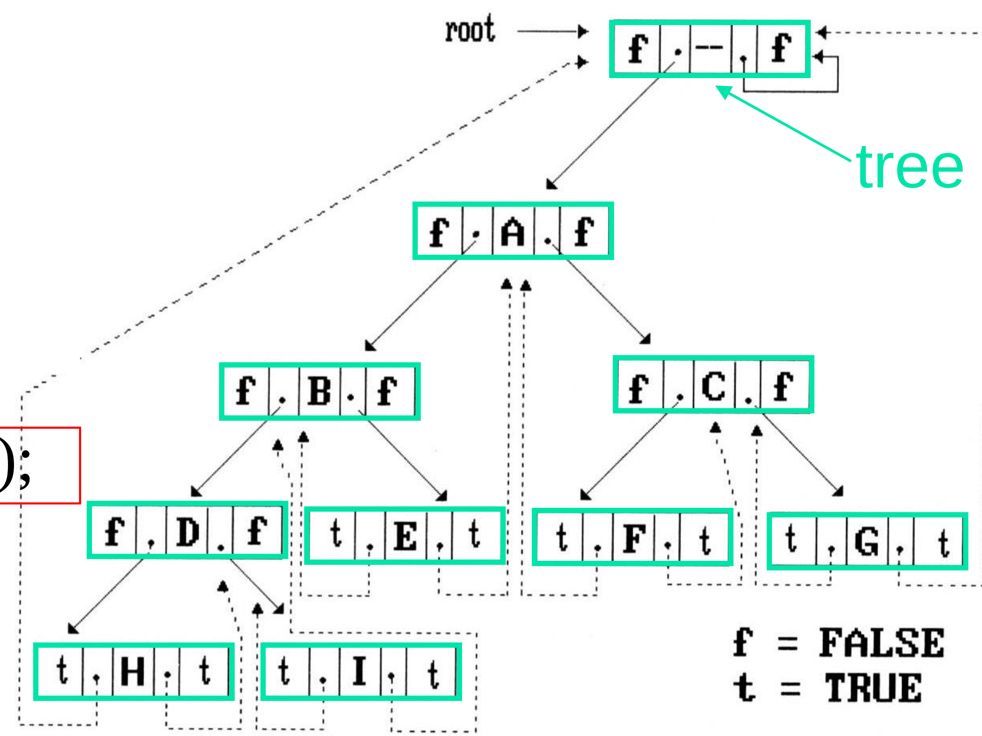


Time Complexity: O(n)

Inorder traversal of a threaded binary tree

```
void tinorder(threaded_pointer tree){
/* traverse the threaded binary tree inorder */
    threaded_pointer temp = tree;
    for (;;) {
      temp = insucc(temp);
      if (temp==tree)
                break;
      printf("%3c",temp->data);
    }
}
```

output: H D I B E A F C G



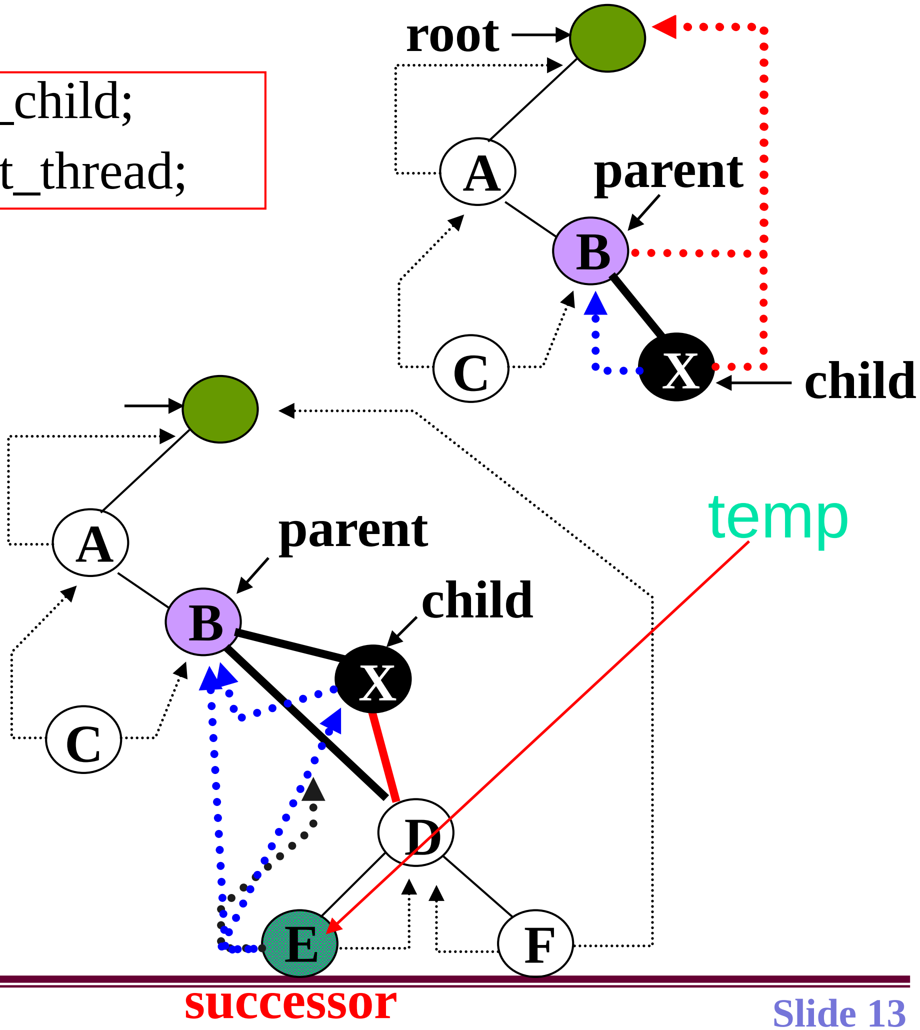
Inserting A Node Into A Threaded Binary Tree

- Insert *child* as the right child of node *parent*

1. change *parent->right_thread* to *FALSE*
2. set *child->left_thread* and *child->right_thread* to *TRUE*
3. set *child->left_child* to point to *parent*
4. set *child->right_child* to *parent->right_child*
5. change *parent->right_child* to point to *child*

Right insertion in a threaded binary tree

```
void insert_right(thread_pointer parent, threaded_pointer child){
/* insert child as the right child of parent in a threaded binary tree */
    threaded_pointer temp;
    child->right_child = parent->right_child;
    child->right_thread = parent->right_thread;
    child->left_child = parent;
    child->left_thread = TRUE;
    parent->right_child = child;
    parent->right_thread = FALSE;
    If(!child->right_thread){
      temp = insucc(child);
      temp->left_child = child;
    }
}
```



Hight Balanced Tree (AVL – Tree)

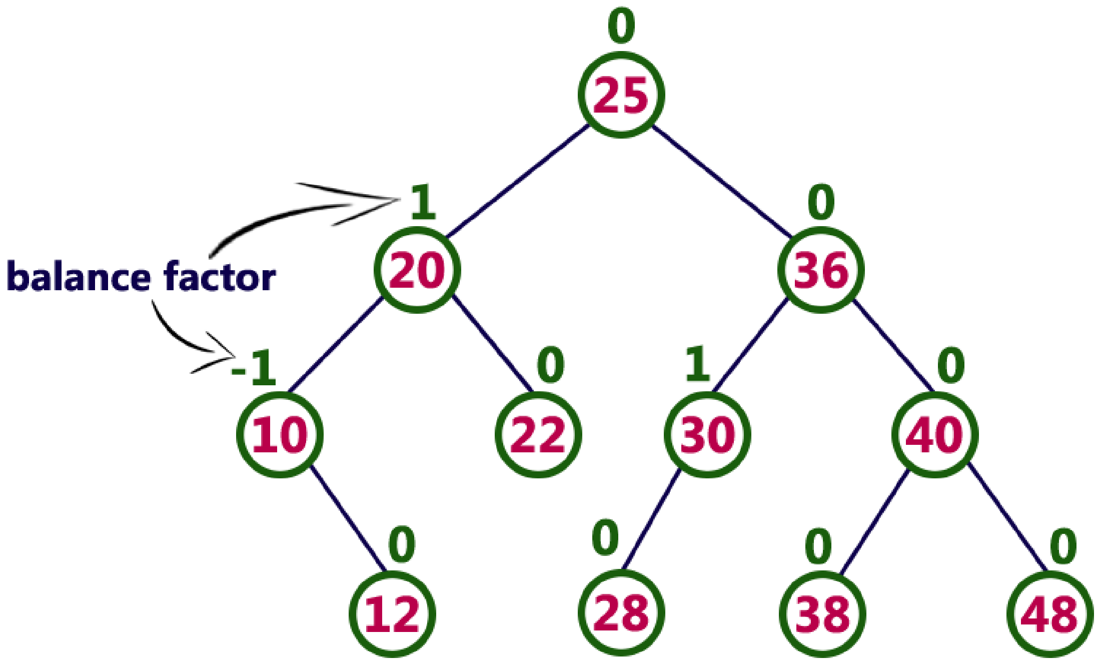
AVL Tree

- AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree.
- A binary tree is said to be balanced, if the difference between the heights of left and right subtrees of every node in the tree is either -1, 0 or +1.
- In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one.
- In an AVL tree, every node maintains a extra information known as **balance factor**.
- The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

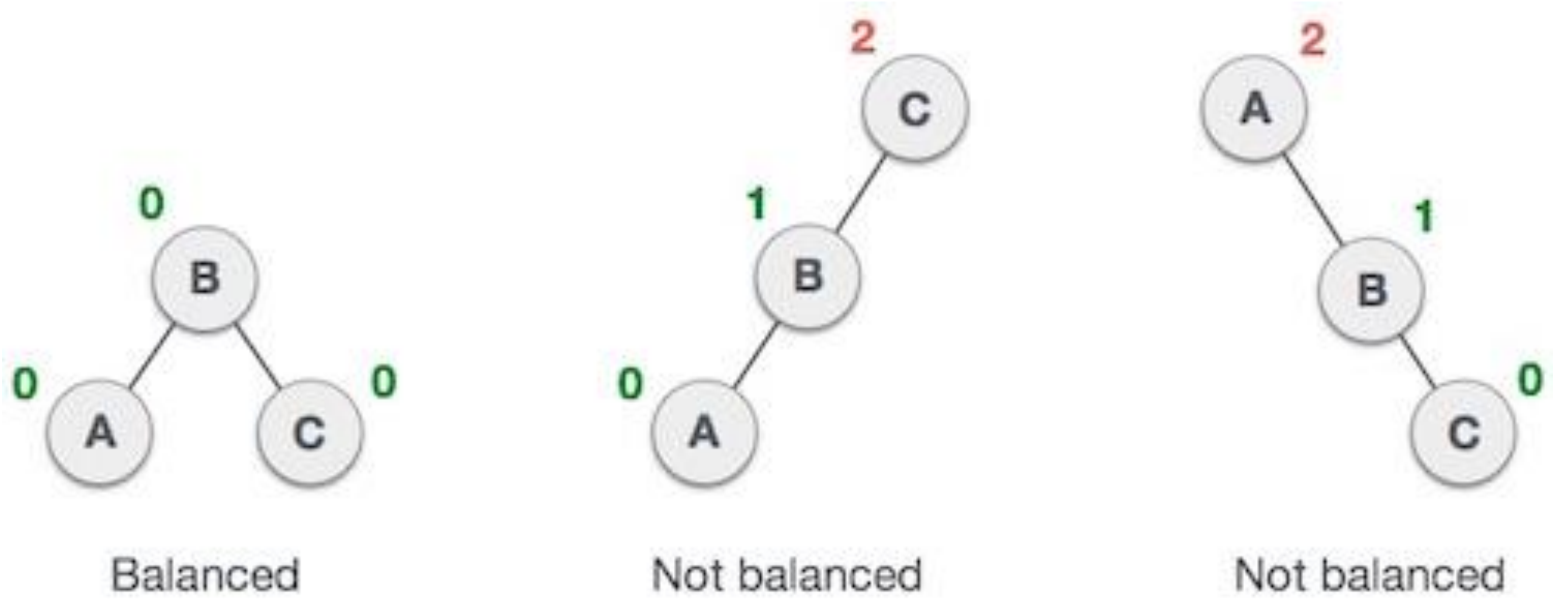
An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

- Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**. In the following explanation, we are calculating as follows...

Balance factor = height_of_Left_Subtree – height_of_Right_Subtree



- The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.
- **Every AVL Tree is a binary search tree but all the Binary Search Trees need not to be AVL trees.**

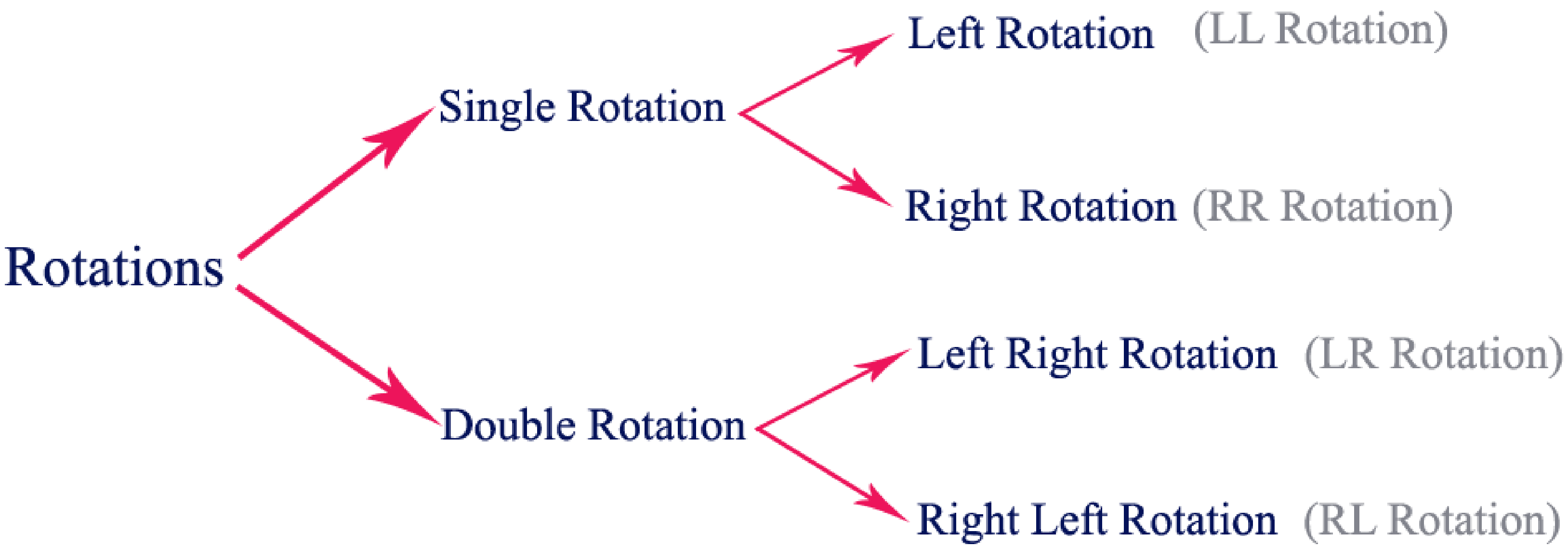

0
B
0
A
C
0
Balanced
2
C
1
B
0
A
Not balanced
2
A
1
B
C
0
Not balanced

AVL Tree Rotations

- In AVL tree, after performing every operation like insertion and deletion we need to check the **balance factor** of every node in the tree.
- If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. We use **rotation** operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.
- Rotation operations are used to make a tree balanced.

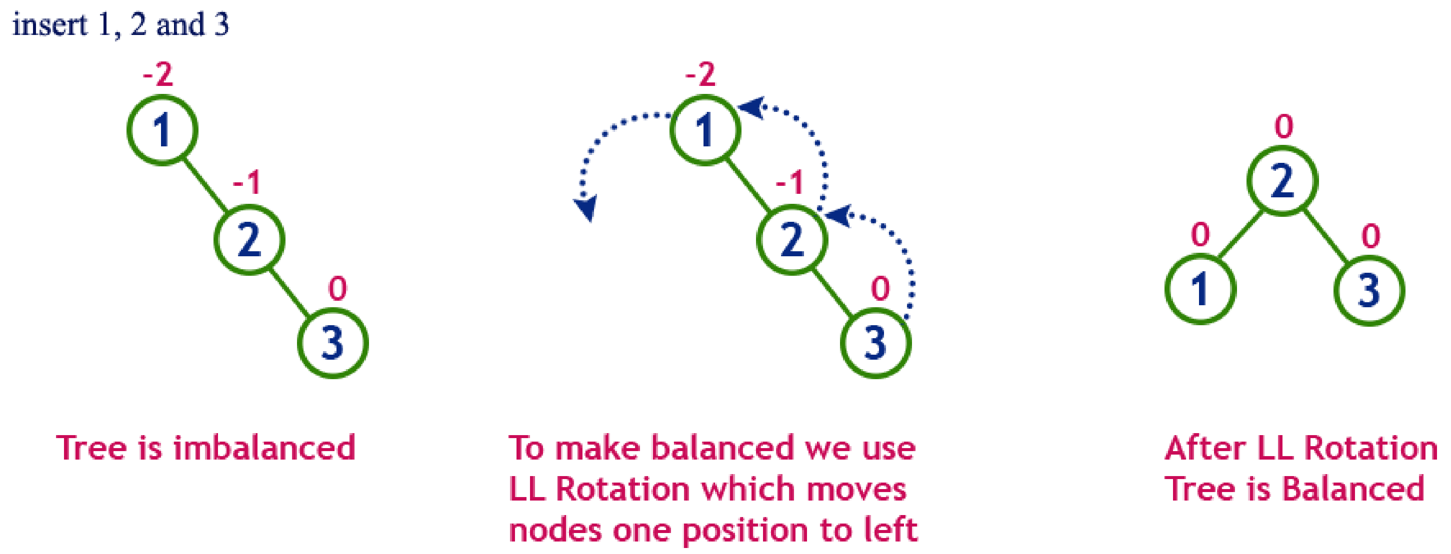
- **Rotation is the process of moving the nodes to either left or right to make tree balanced.**

- There are **four** rotations and they are classified into **two** types.



Single Left Rotation (LL Rotation)

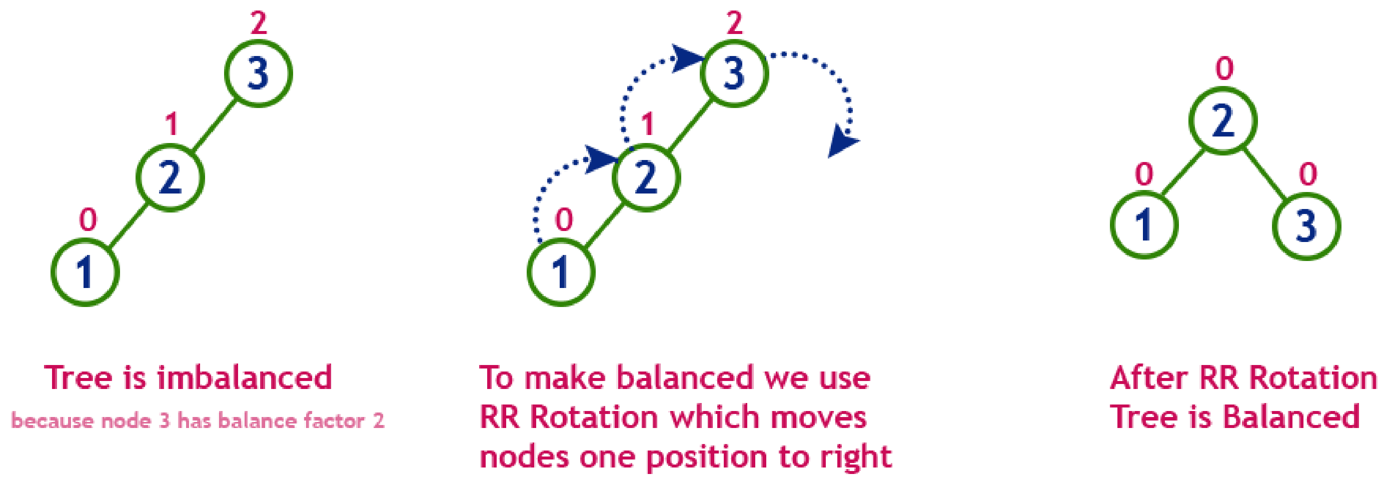
- In LL Rotation every node moves one position to left from the current position.
- To understand LL Rotation, let us consider following insertion operations into an AVL Tree...



Single Right Rotation (RR Rotation)

- In RR Rotation every node moves one position to right from the current position.
- To understand RR Rotation, let us consider following insertion operations into an AVL Tree...

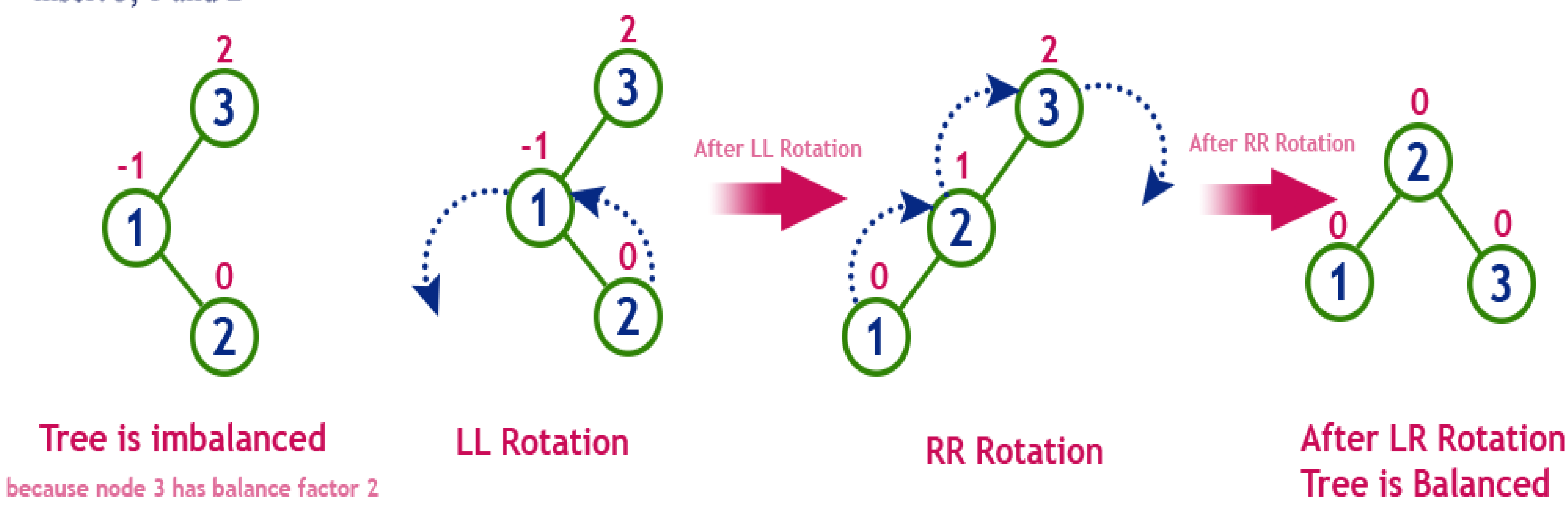
insert 3, 2 and 1



Left Right Rotation (LR Rotation)

- Double rotations are slightly complex version of already explained versions of rotations.
- The LR Rotation is combination of single left rotation followed by single right rotation.
- In LR Rotation, first every node moves one position to left then one position to right from the current position.
- To understand LR Rotation, let us consider following insertion operations into an AVL Tree...

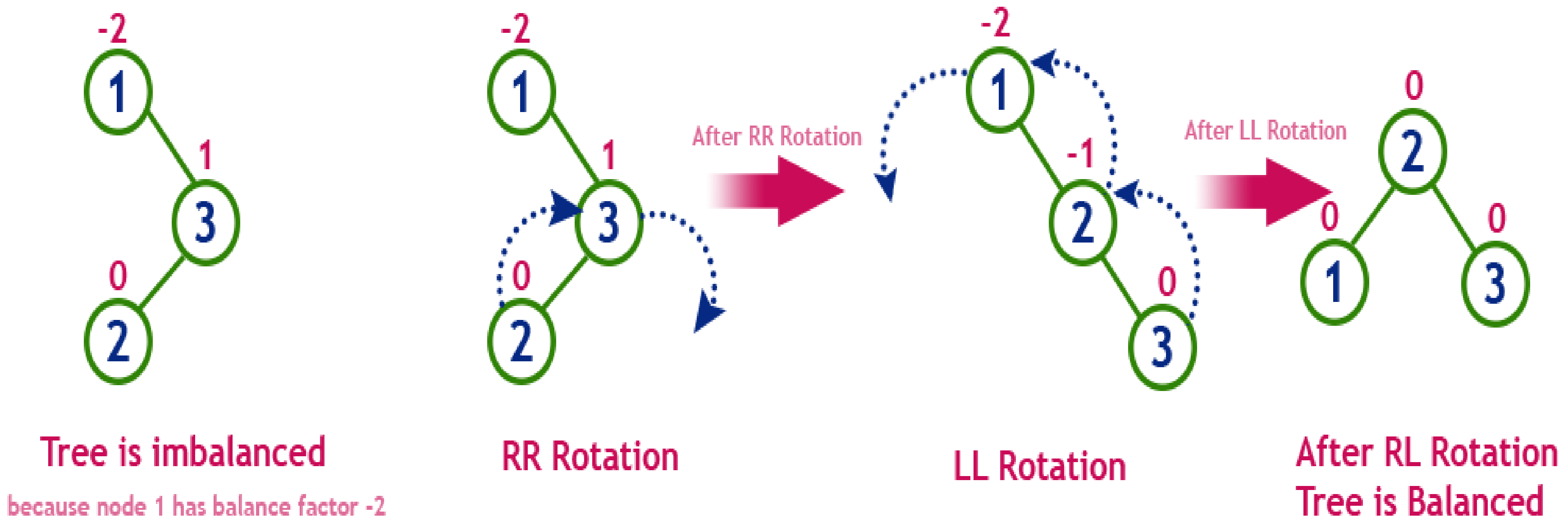
insert 3, 1 and 2



Right Left Rotation (RL Rotation)

- The RL Rotation is combination of single right rotation followed by single left rotation.
- In RL Rotation, first every node moves one position to right then one position to left from the current position.
- To understand RL Rotation, let us consider following insertion operations into an AVL Tree...

insert 1, 3 and 2



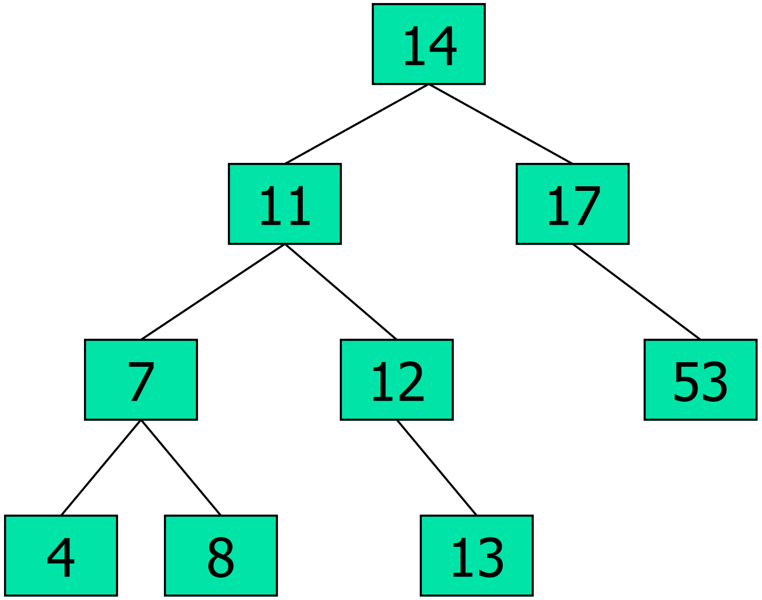
AVL Tree Demonstration

Operations on an AVL Tree

- The following operations are performed on an AVL tree...
  - **Search**
  - **Insertion**
  - **Deletion**
  - **Traversal**

Search Operation in AVL Tree

- In an AVL tree, the search operation is performed with **O(log n)** time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree...
  - **Step 1:** Read the search element from the user
  - **Step 2:** Compare, the search element with the value of root node in the tree.
  - **Step 3:** If both are matching, then display "Given node found!!!" and terminate the function
  - **Step 4:** If both are not matching, then check whether search element is smaller or larger than that node value.
  - **Step 5:** If search element is smaller, then continue the search process in left subtree.
  - **Step 6:** If search element is larger, then continue the search process in right subtree.
  - **Step 7:** Repeat the same until we found exact element or we completed with a leaf node
  - **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.
  - **Step 9:** If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.


14
11
17
7
12
53
4
8
13

Insertion Operation in AVL Tree

- In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...
  - **Step 1:** Insert the new element into the tree using Binary Search Tree insertion logic.
  - **Step 2:** After insertion, check the **Balance Factor** of every node.
  - **Step 3:** If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.
  - **Step 4:** If the **Balance Factor** of any node is other than **0 or 1 or -1** then tree is said to be imbalanced. Then perform the suitable **Rotation** to make it balanced. And go for next operation.

- **Construct an AVL Tree by inserting numbers from 1 to 8**

Deletion Operation in AVL Tree

- In an AVL Tree, the deletion operation is similar to deletion operation in BST. But after every deletion operation we need to check with the Balance Factor condition.

- If the tree is balanced after deletion then go for next operation otherwise perform the suitable rotation to make the tree Balanced.