5 : Trees

IT 3206– Data Structures and Algorithms

Level II - Semester 3

Overview

- In this chapter, you will learn about a non-linear data structure called tree.
- A tree is a structure which is mainly used to store data that is hierarchical in nature.
- Further, from this chapter you will learn the followings :
    - Introduction to trees.
    - Different types of selected trees and their application
    - Implementation of different trees

Intended Learning Outcomes

- At the end of this lesson, you will be able to;
  - Explain tree structures and properties.
  - Demonstrate implementation of trees, tree traversal techniques and tree balancing techniques.
  - Explain the usage of a heap

List of subtopics

5.1 Introduction to trees

- 5.1.1 Introduction to general trees, properties of general trees
- 5.1.2 Introduction to binary trees, properties of binary trees

5.2 Tree traversal

- 5.2.1 Depth First Traversal
  - 5.2.1.1 Preorder Traversal
  - 5.2.1.2 Inorder Traversal
  - 5.2.1.3 Postorder Traversal
- 5.2.2 Breadth First Traversal (Level Order Traversal)

5.3 Binary search trees

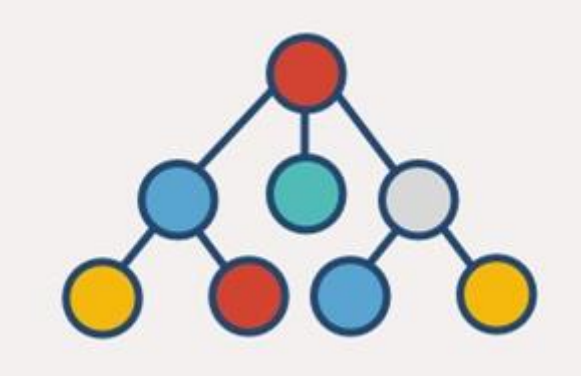
5.4 Tree Balancing

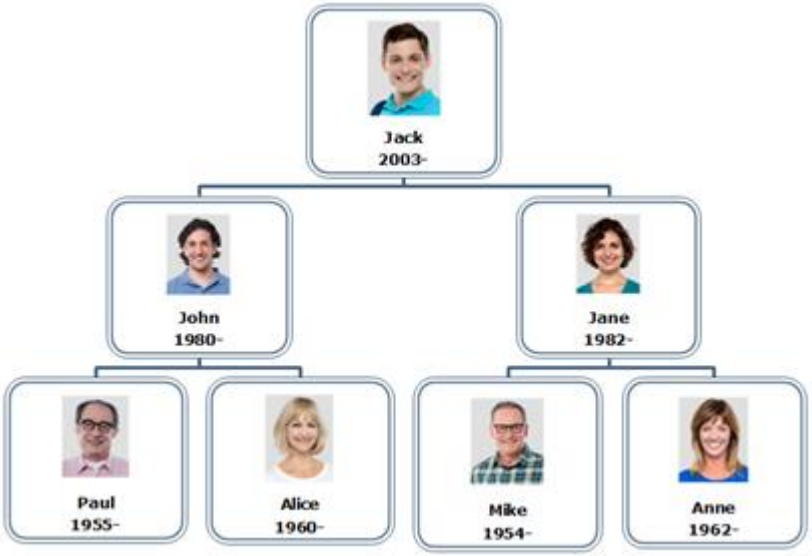
- 5.4.1 Introduction to tree balancing
- 5.4.2 Global tree balancing (The DSW algorithm)
- 5.4.3 Local tree balancing (AVL Tree)

5.5 Heaps

5.1 Introduction to trees

- A tree is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to a number of nodes.
- Tree is an example of a nonlinear data structure.
- A *tree* structure is a way of representing the hierarchical nature of a structure in a graphical form.(think of a *family tree*, where the children are grouped under their parents in the tree)





5.1 Introduction to trees

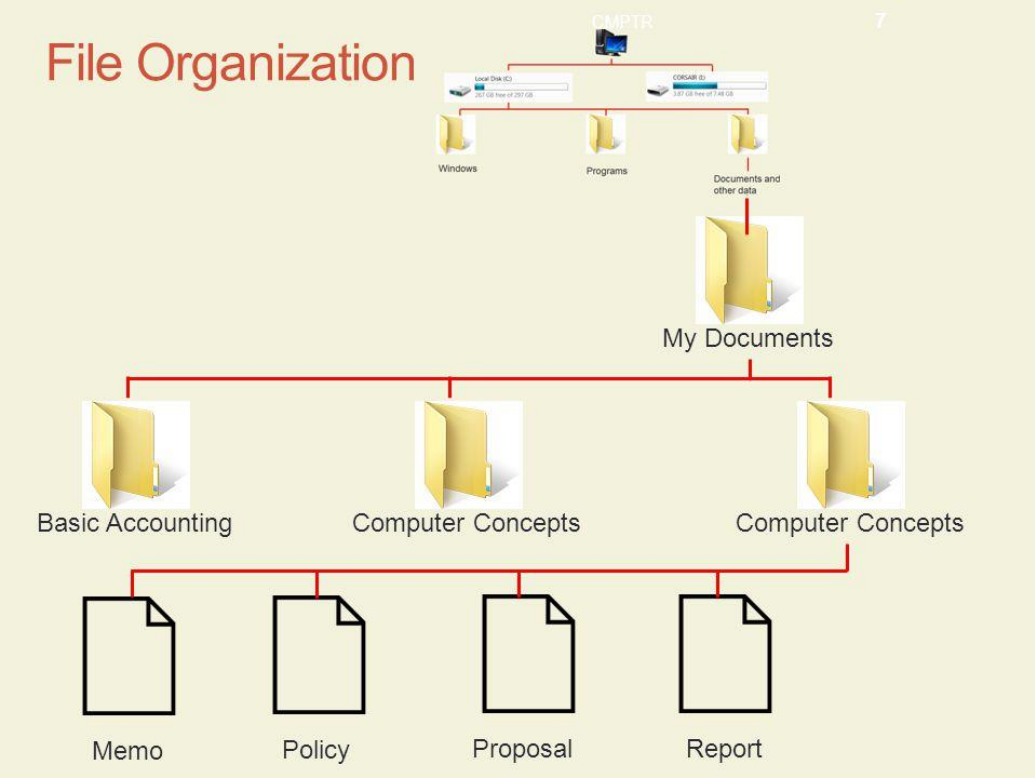
- Tree is one of the most important non-linear data structures in computing.
- Different types of trees discuss in this chapter.
  - General tree — each node can have an arbitrary number of children.
  - Binary tree — each node has at most two children
  - Binary search trees
  - AVL trees
- It allows us to implement faster algorithms(compared with algorithms using linear data structures).
  - Mostly all operating systems store files in trees or tree like structures.
  - Compilers use a syntax tree to validate the syntax of every program you write
- Practical usage in trees
  - Mostly all operating systems store files in trees or tree like structures.
  - Compilers use a syntax tree to validate the syntax of every program you write

5.1 Introduction to trees

- **Features and application of trees**
    - A binary tree is a type of data structure for storing data such as numbers in an organized way.
    - Binary search trees allow binary search for fast lookup, addition and removal of data items, and can be used to implement dynamic sets and lookup tables
    - Directory structure in many common operating systems, including Unix, DOS, and windows.
    - AVL trees are mostly used for in-memory sorts of sets and dictionaries.
    - AVL trees are also used extensively in database applications in which insertions and deletions are fewer but there are frequent lookups for data required.

5.1 Introduction to trees

Example for a general tree:



5.1.1 Introduction to general trees, properties of general trees

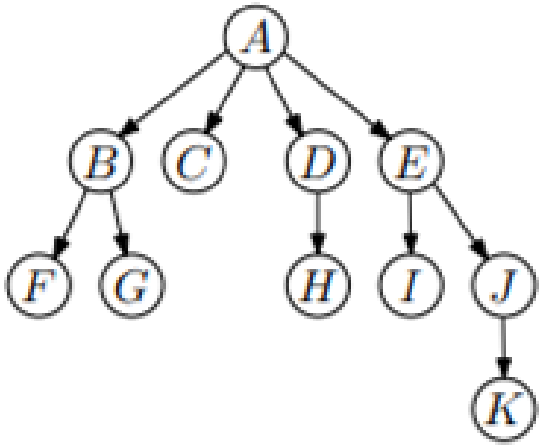
- A tree consists of a set of nodes and a set of directed edges that connect pairs of nodes.
- A tree has the following properties.
  - One node is distinguished as the root.
  - Every node c, except the root, is connected by an edge from exactly one other node p. Node p is c's parent, and c is one of p's children.
  - A unique path traverses from the root to each node. The number of edges that must be followed is the path length.
- Parents and children are naturally defined.
- A directed edge connects the parent to the child.
- Trees can be defined in two ways :
  - Recursive
  - Non- recursive

  Implementation of the tree is more convenient in a recursive approach than a non-recursive approach.

5.1.1 Introduction to general trees, properties of general trees

Non-recursive definition:

- A (rooted) tree consists of a set of nodes, and a set of directed edges between nodes.
- One node is the root; For every node c that is not the root, there is exactly one edge (p, c) pointing to c;
- For every node c there is a unique path from the root to c.



5.1.1 Introduction to general trees, properties of general trees

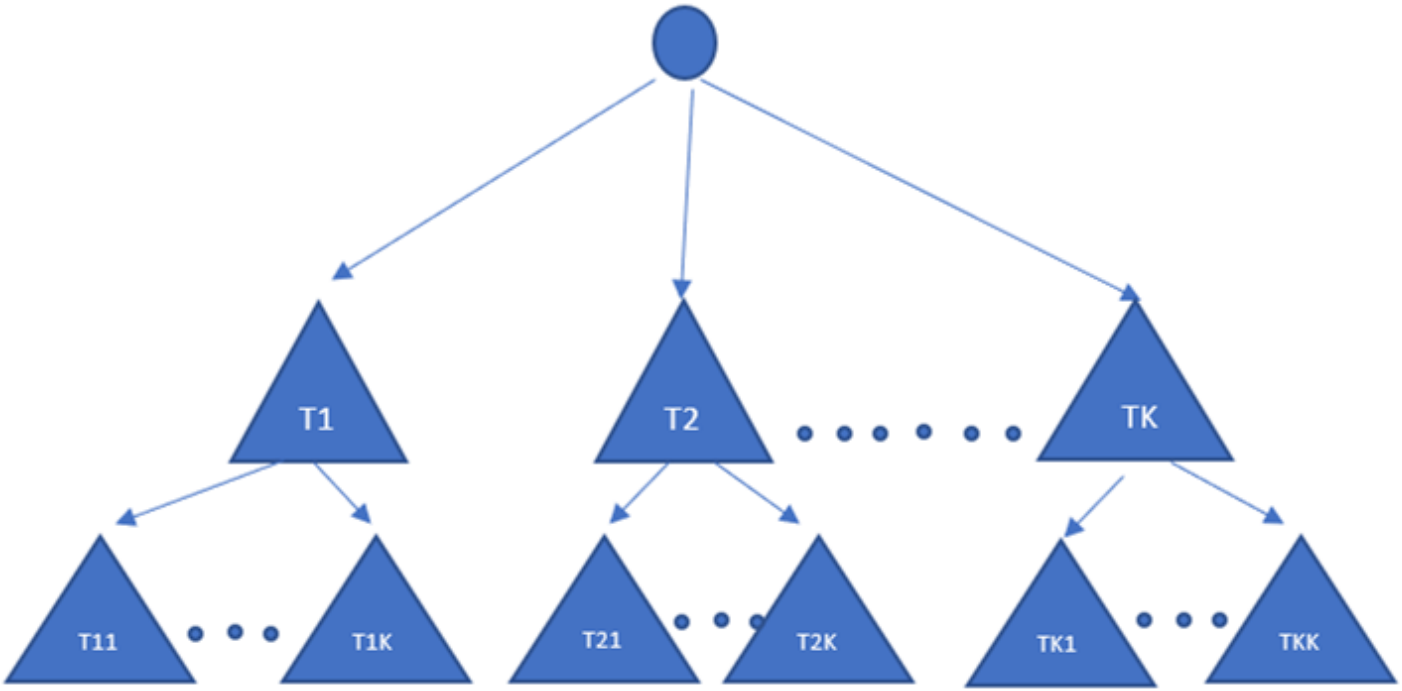
General trees - definition:

- A tree is a collection of nodes. The collection can be empty; otherwise a tree consists of a distinguish node n, called root, and zero or more non-empty (sub)trees T1,T2,T3,….Tn. . Each of whose roots are connected by a directed edge from n.

5.1.1 Introduction to general trees, properties of general trees

Recursive definition:

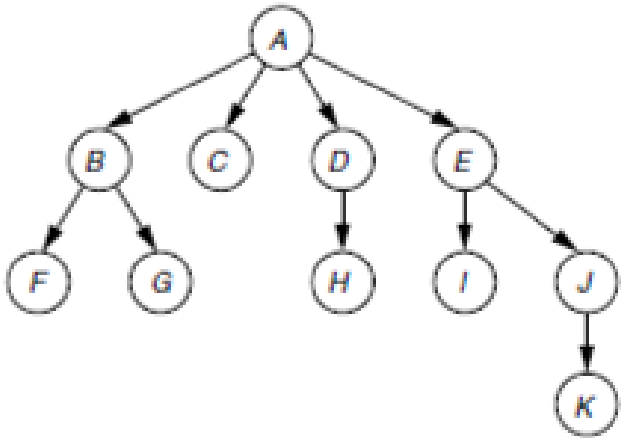
Example:



5.1.1 Introduction to general trees, properties of general trees

A tree is a collection of nodes. The collection can be empty; otherwise a tree consists of a distinguish node r, called root, and zero or more nonempty (sub)trees $T_1, T_2, T_3, \ldots T_K$. Each of whose roots are connected by a directed edge from r.

- The root node is A.
- A's children are B, C, D, and E.
- Since A is the root, it has no parent; all other nodes have parents.
- For instance, *B*'s parent is *A*.
- A node that has no children is called a *leaf.*
- The leaves in this tree are *C, F, G, H, I,* and *K*.
- The length of the path from *A* to *K* is 3 (edges)
- The length of the path from A to A is 0 (edges).



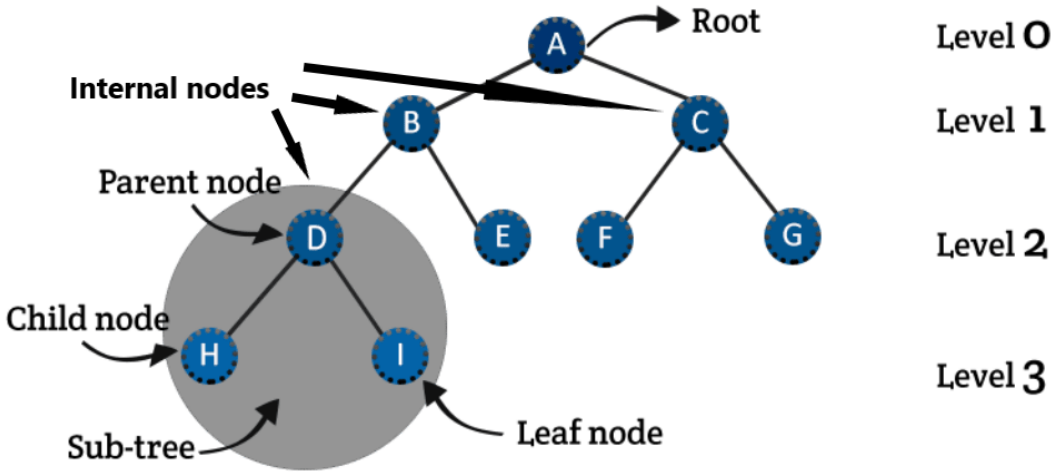
5.1.1 Introduction to general trees, properties of general trees

- **Basic Terminology**
  - **tree** - a non-empty collection of vertices & edges
  - **vertex (node)** - can have a name and carry other associated information
  - **path** - list of distinct vertices in which successive vertices are connected by edges
  - **Sub-trees** -If the root node R is not NULL, then the trees T1, T2, and T3 are called the sub-trees of R.
  - **Leaf node -**A node that has no children is called the leaf node or the terminal node.
  - **Non-leaf node-**In any tree the node which has at least one child is called internal/non-leaf node.

5.1.1 Introduction to general trees, properties of general trees

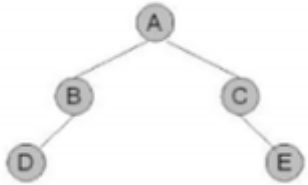
- **Basic Terminology**
  - **Level number** Every node in the tree is assigned a level number in such a way that the root node is at level 0, children of the root node are at level number 1.
  - Thus, every node is at one level higher than its parent. So, all child nodes have a level number given by parent's level number + 1.

Example:



5.1.1 Introduction to general trees, properties of general trees

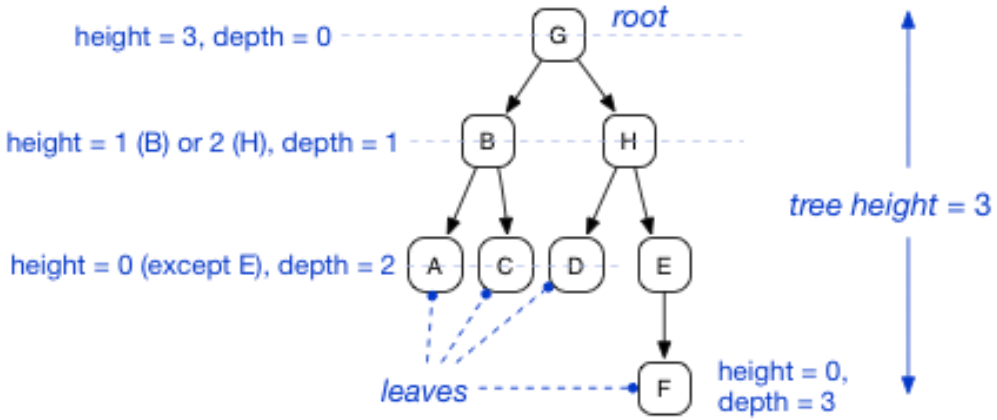
- **Basic Terminology**
  - **Degree**: Degree of a node is equal to the number of children that a node has.
  - The degree of a leaf node is zero.
  - **In-degree**: In-degree of a node is the number of edges arriving at that node. For example, in degree of the node B is one i.e. one edge merges.
  - **Out-degree**: Out-degree of a node is the number of edges leaving that node. For example, out degree of the node A is two i.e. two edges comes out of this root node.



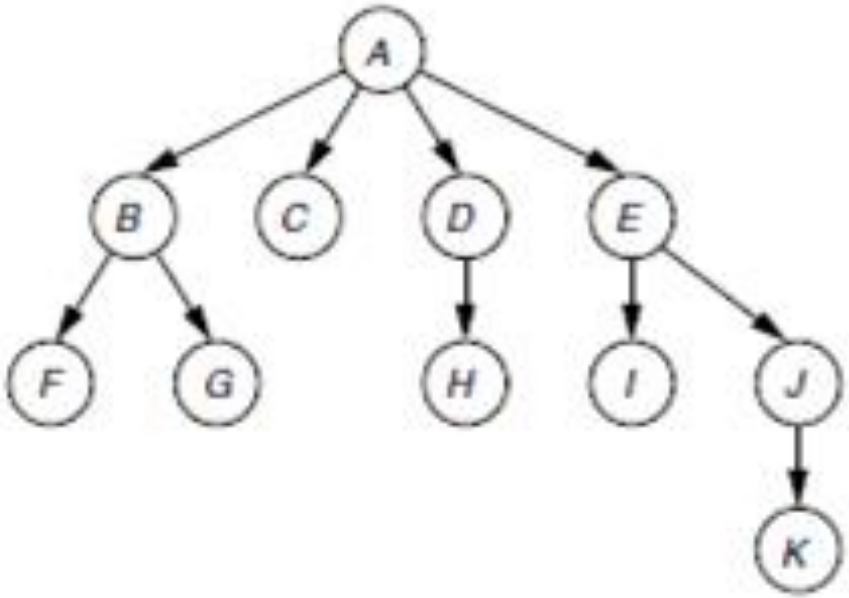
5.1.1 Introduction to general trees, properties of general trees

- A tree with N nodes must have N – 1 edges because every node except the parent has an incoming edge.
- The **depth** of a node is the length of the path from the root to the node.
- The **height** of a node is the length of the path from the node to the deepest leaf.(Thus the height of E is 2)
- The height of any node is 1 more than the height of its maximum-height child. Thus the height of a tree is the height of the root.

Example:



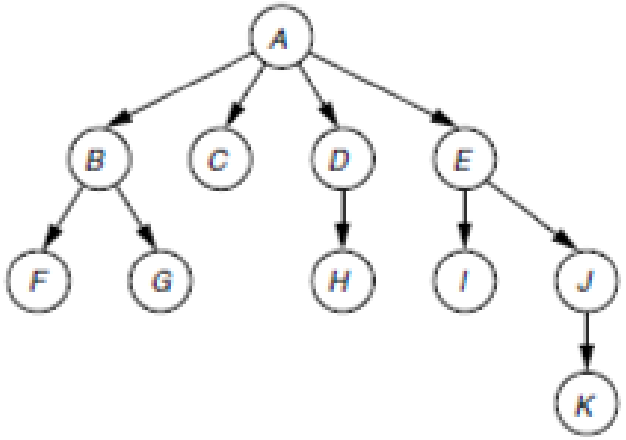
5.1.1 Introduction to general trees, properties of general trees



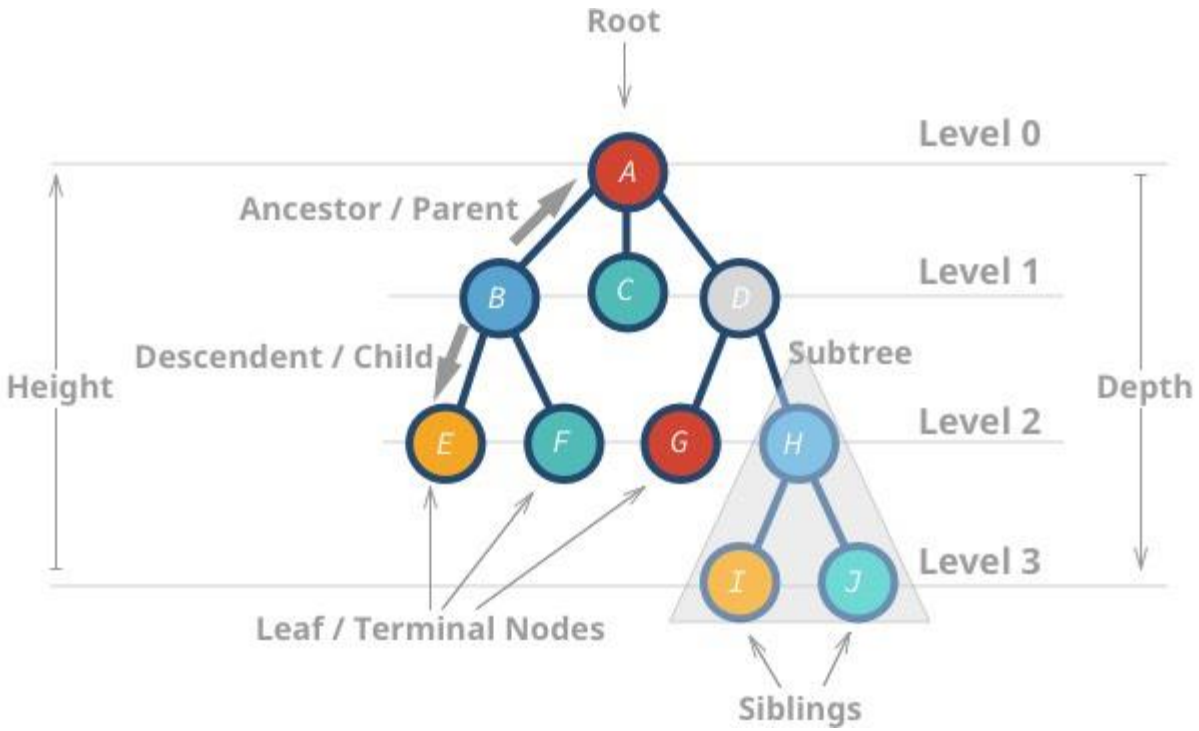
Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

5.1.1 Introduction to general trees, properties of general trees

- Nodes with the same parent are called *siblings;* thus *B, C, D,* and *E* are all siblings
- If there is a path from node *u* to node *v,* then *u* is an *ancestor* of *v* and *v* is a *descendant* of *u*.
- If $u \neq v$, then *u* is a *proper ancestor* of *v* and *v* is a *proper descendant* of *u*.
- The *size of a node* is the number of descendants the node has (including the node itself). Thus the size of *B* is 3, and the size of *C* is 1.
- The size of a tree is the size of the root. Thus the size of the tree shown in Figure is the size of its root *A,* or 11.



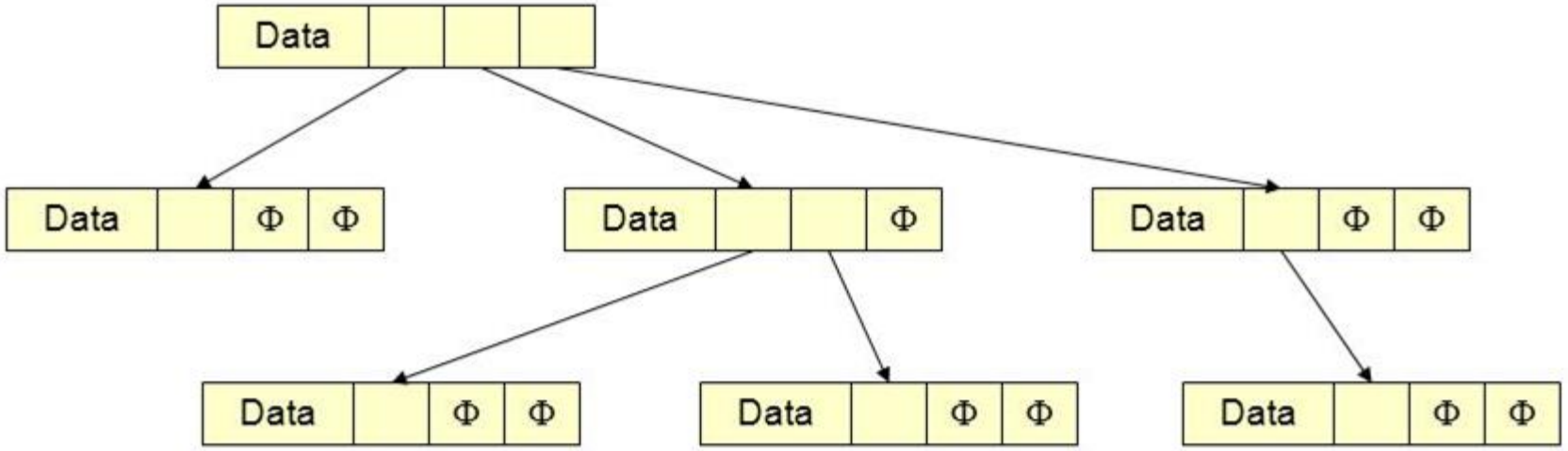
5.1.1 Introduction to general trees, properties of general trees



5.1.1 Introduction to general trees, properties of general trees

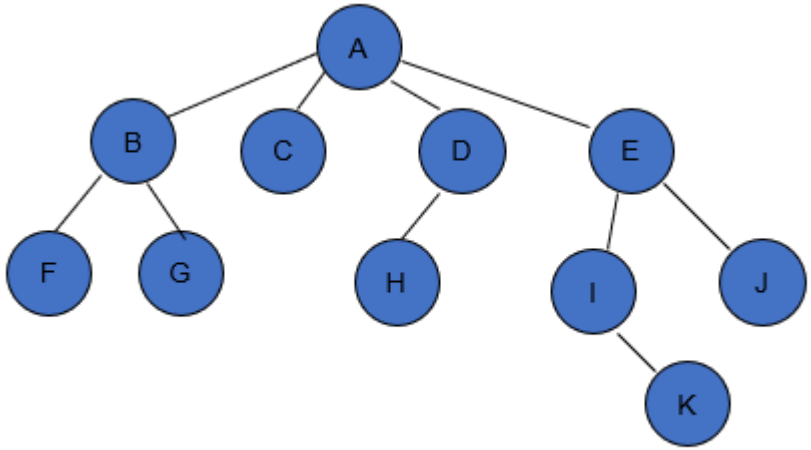
Representation of Trees

- Every tree node:
  - object – useful information
  - children – pointers to its children nodes



5.1.1 Introduction to general trees, properties of general trees

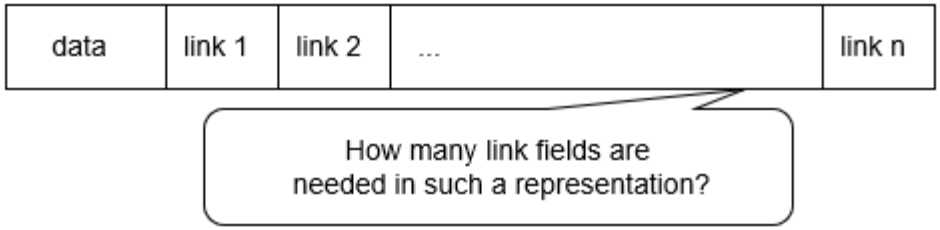
- A tree consists of set of nodes and set of edges that connected pair of nodes.



5.1.1 Introduction to general trees, properties of general trees

List Representation of trees

- (A (B (E (K, L), F), C (G), D (H (M), I, J)))
- How many link fields are needed to represent the tree?
- The root comes first, followed by a list of sub-trees



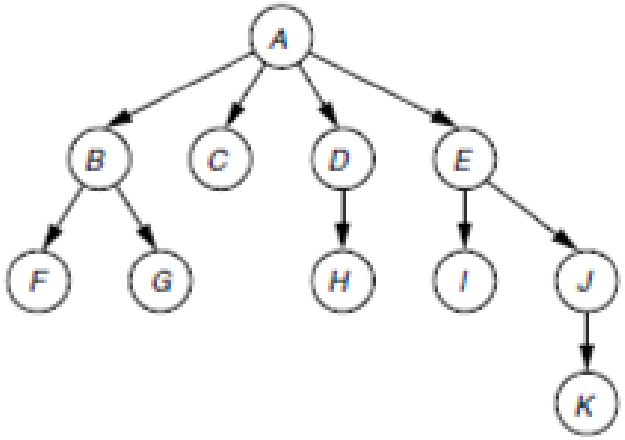
- In worst case we need n links and for best case, we need 2 links

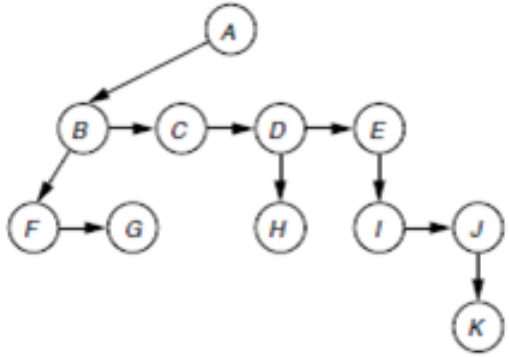
5.1.1 Introduction to general trees, properties of general trees

- Implementation
  - One way to implement a tree would be to have in each node a link to each child of the node in addition to its data.
  - However, as the number of children per node can vary greatly and is not known in advance, making the children direct links in the data structure might not be feasible.
  - There would be too much wasted space.
  - The solution is called the *first child/next sibling* method

5.1.1 Introduction to general trees, properties of general trees

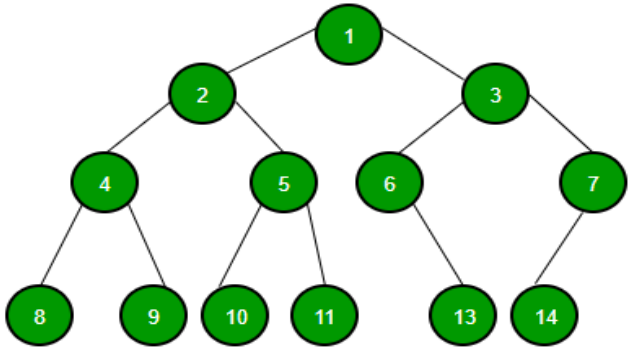
- first child/next sibling method
  - Keep the children of each node in a linked list of tree nodes, with each node keeping two links, one to its leftmost child (if it is not a leaf) and one to its right sibling (if it is not the rightmost sibling).





5.1.2 Introduction to binary trees, properties of binary trees

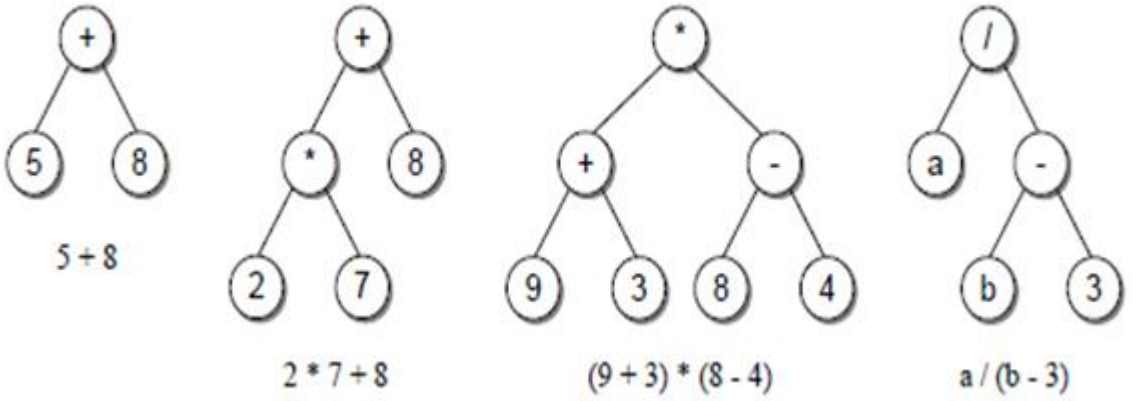
- A binary tree is a tree in which no node can have more than two children.
- The two children of each node in a binary tree are called the left child and the right child.
- A node in a binary tree doesn't necessarily have the maximum of two children; it may have only a left child, or only a right child, or it can have no children at all (in which case it's a leaf).



5.1.2 Introduction to binary trees, properties of binary trees

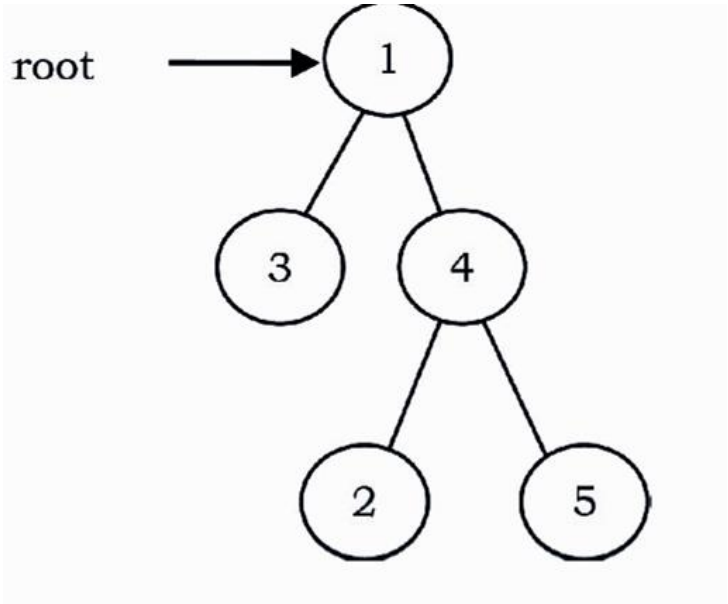
- An expression tree is one example of the use of binary trees. Such trees are central data structures in compiler design.
- A second use of the binary tree is the Huffman coding tree, which is used to implement a simple but relatively effective data compression algorithm.
- Each symbol in the alphabet is stored at a leaf. Its code is obtained by following the path to it from the root.

Examples



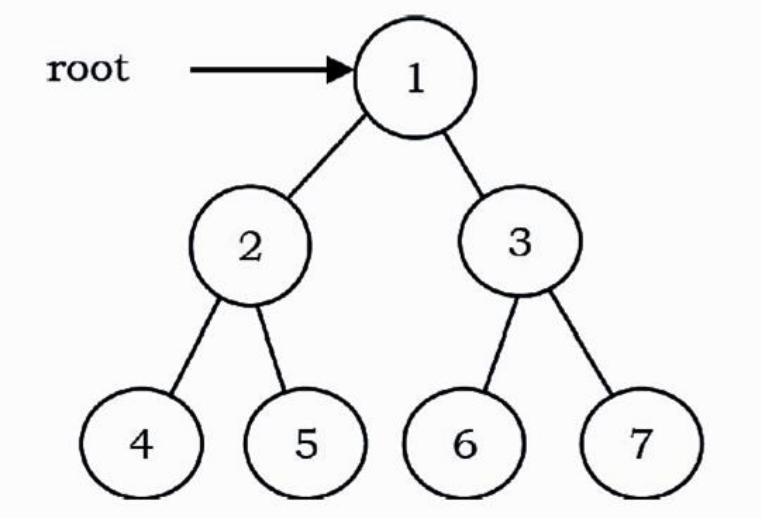
5.1.2 Introduction to binary trees, properties of binary trees

- Types of Binary Trees
  - **Strict Binary Tree**: A binary tree is called strict binary tree if each node has exactly two children or no children.



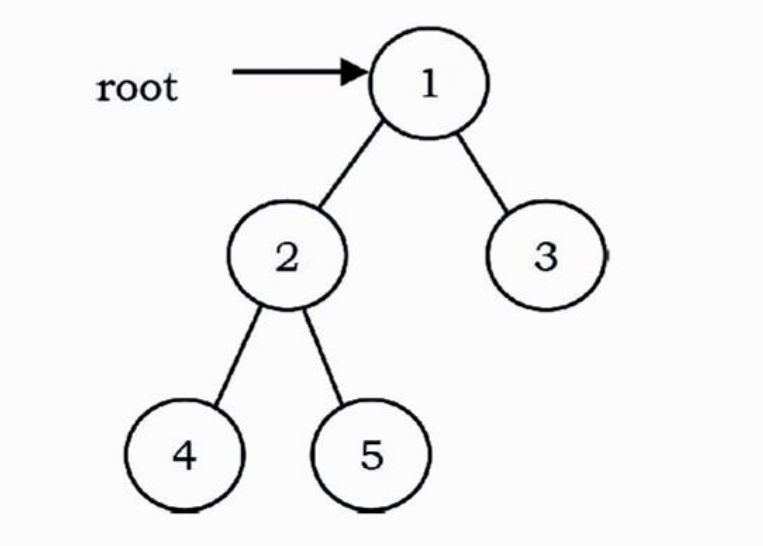
5.1.2 Introduction to binary trees, properties of binary trees

- **Full Binary Tree:** A binary tree is called full binary tree if each node has exactly two children and all leaf nodes are at the same level**.**



5.1.2 Introduction to binary trees, properties of binary trees

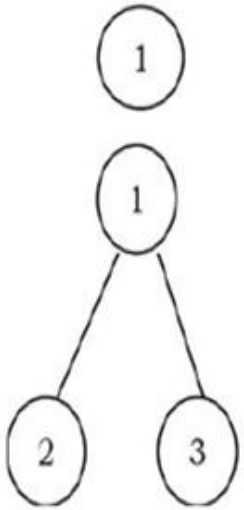
- **Complete Binary Tree:** A binary tree is called complete binary tree if all leaf nodes are at height h or h – 1 and also without any missing number in the sequence.



5.1.2 Introduction to binary trees, properties of binary trees

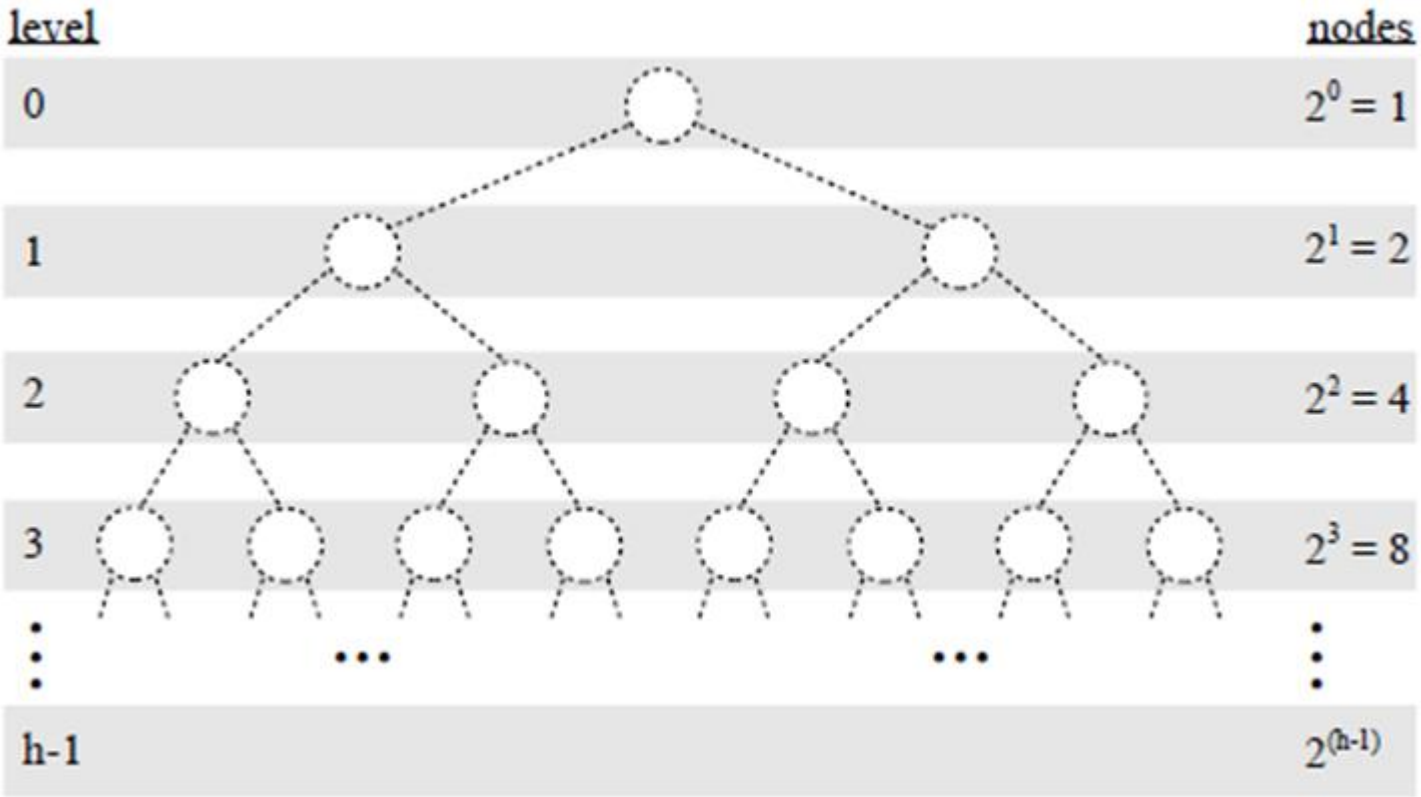
- **Properties of Binary Trees**

Let us assume that the height of the tree is *h*. Also, assume that root node is at height zero.



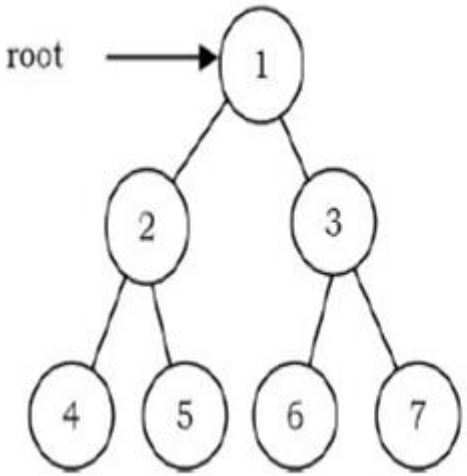
Height	Number of nodes at level *h*
$h = 0$	$2^0 = 1$
$h = 1$	$2^1 = 2$

5.1.2 Introduction to binary trees, properties of binary trees



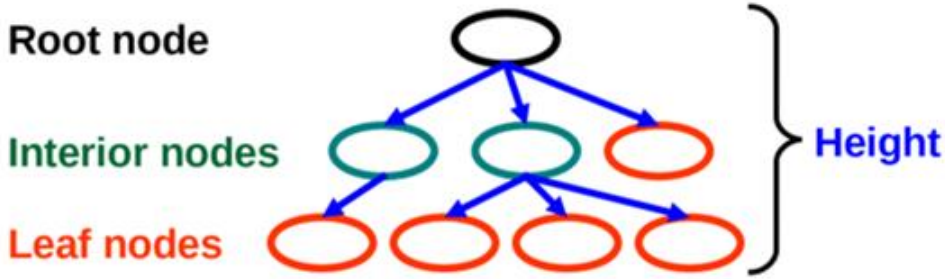
5.1.2 Introduction to binary trees, properties of binary trees

- **Properties of Binary Trees**



$h = 2$ $2^2 = 4$

Root ⇒ no parent
Leaf ⇒ no child
Interior ⇒ non-leaf
Height ⇒ distance from root to leaf



5.1.2 Introduction to binary trees, properties of binary trees

- **Properties of Binary Trees**
  - The number of nodes n in a full binary tree is $2^{h+1} - 1$. Since, there are *h* levels we need to add all nodes at each level [$2^0 + 2^1 + 2^2 + \cdots + 2^h = 2^{h+1} - 1$].
  - The number of nodes *n* in a complete binary tree is between 2^h (minimum) and $2^{h+1} - 1$ (maximum).
  - The number of leaf nodes in a full binary tree is 2^h.
  - The number of NULL links (wasted pointers) in a complete binary tree of n nodes is $n + 1$.

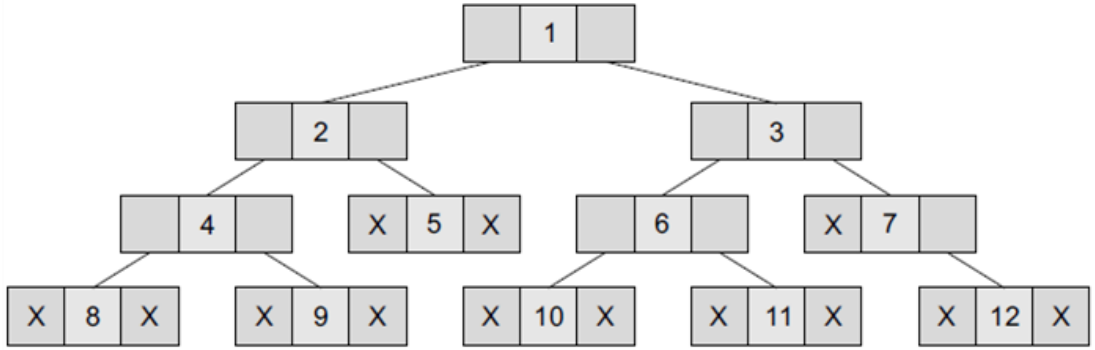
5.1.2 Introduction to binary trees, properties of binary trees

- **Properties of Binary Trees**

	At least	At most
The number of leaf nodes	$h+1$	2^h
The number of non-leaf nodes	h	$2^h - 1$
The total number of nodes	$2h+1$	$2^{h+1} - 1$
The height of the tree h	$\log(n+1) - 1$	$(n-1)/2$

5.1.2 Introduction to binary trees, properties of binary trees

- **Representation of Binary Trees in the Memory**
  - In the computer's memory, a binary tree can be maintained either by using a linked representation or by using a sequential representation.
  - In the linked representation of a binary tree, every node will have three parts: the data element, a pointer to the left node, and a pointer to the right node.

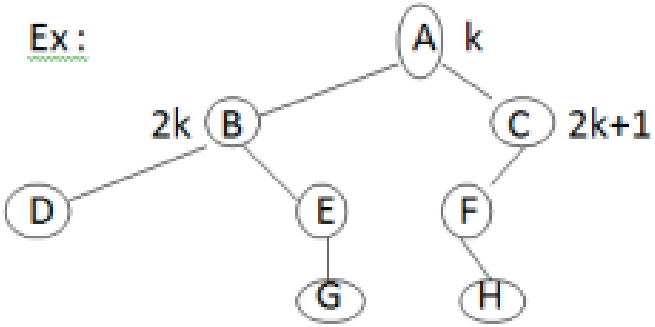


5.1.2 Introduction to binary trees, properties of binary trees

- **Representation of Binary Trees in the Memory**
  - Sequential representation of trees is done using single or one-dimensional arrays.
  - A one-dimensional array, called TREE, is used to store the elements of tree.
  - The root of the tree will be stored in the first location. That is, TREE[1] will store the data of the root element.
  - The children of a node stored in location K will be stored in locations (2 × K) and (2 × K+1).
  - The maximum size of the array TREE is given as (2h–1), where h is the height of the tree.
  - An empty tree or sub-tree is specified using NULL. If TREE[1] = NULL, then the tree is empty.

5.1.2 Introduction to binary trees, properties of binary trees

- **Sequential representation of Binary Tree**



TREE

A	B	C	D	E	F					G		H		
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

5.1.2 Introduction to binary trees, properties of binary trees

- **Representation of Binary Trees in the Memory (pointer based)**

In the computer's memory, a binary tree can be maintained either by using a linked representation or by using a sequential representation.

- **Linked representation of binary trees**
  - In the linked representation of a binary tree, every node will have three parts:
    - the data element,
    - a pointer to the left node, and
    - a pointer to the right node.

•So in JAVA, the binary tree is built with a node type given below

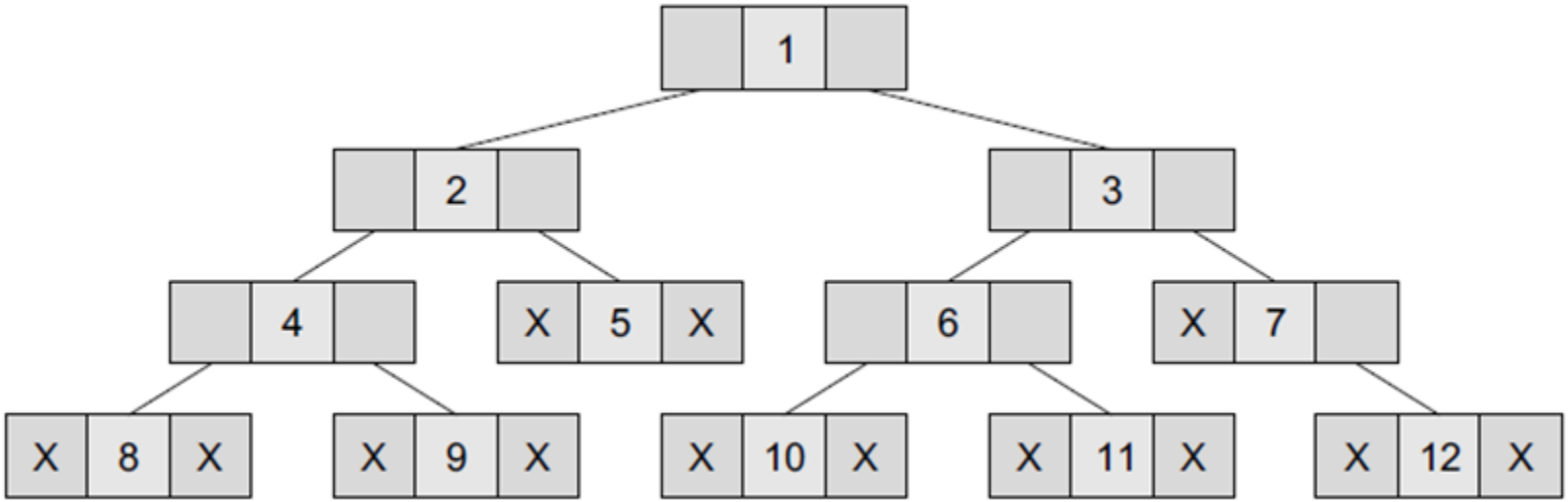
5.1.2 Introduction to binary trees, properties of binary trees

- **Implementation of Binary trees**

```
class Node
{
    int key;
    Node left, right;

    public Node(int item)
    {
        key = item;
        left = right = null;
    }
}
```

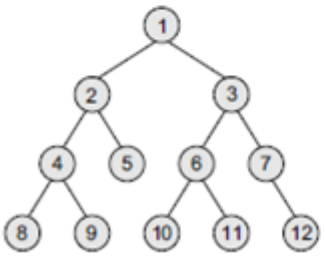
5.1.2 Introduction to binary trees, properties of binary trees

- **Representation of Binary Trees in the Memory (pointer based)**



5.1.2 Introduction to binary trees, properties of binary trees

The above tree is represented in the main memory using a linked list as follows:



ROOT 3

AVAIL 15

	LEFT	DATA	RIGHT
1	–1	8	–1
2	–1	10	–1
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12
10			
11	–1	7	18
12	–1	9	–1
13			
14	–1	5	–1
15			
16	–1	11	–1
17			
18	–1	12	–1
19			
20	2	6	16

5.1.2 Introduction to binary trees, properties of binary trees

- **Implementation of Binary trees**

  There are other approaches to designing class Node. Instead of placing the data items directly into the node, you could use a reference to an object representing the data item:

```
class Node
{
    person p1; // reference to person object
    node leftChild; // this node's left child
    node rightChild; // this node's right child
}
class person
{
    int iData;
    float fData;
}
```

5.2 Tree traversal

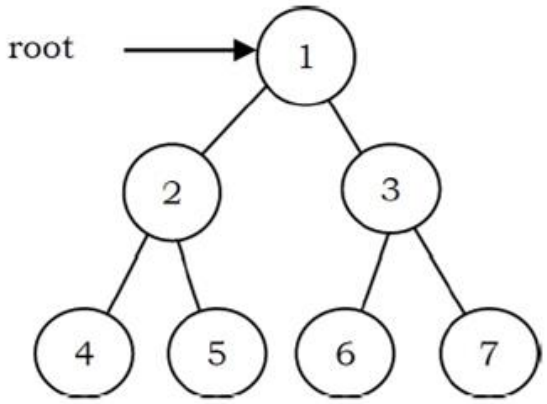
- The process of visiting all nodes of a tree is called tree traversal.
- Each node is processed only once but it may be visited more than once.
- As we have already seen in linear data structures (like linked lists, stacks, queues, etc.), the elements are visited in sequential order. But, in tree structures there are many different ways.
- Tree traversal is like searching the tree, except that in traversal the goal is to move through the tree in a particular order.
- In addition, all nodes are processed in the traversal but searching stops when the required node is found.

5.2.1 Depth First Traversal

- There are three basic traversal mechanisms under the Depth First Traversal :
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal

5.2.1 Depth First Traversal

- Pre-order traversal
  - In preorder traversal, each node is processed before (pre) either of its sub trees.
  - Even though each node is processed before the sub trees, it still requires that some information must be maintained while moving down the tree.
  - In the figure below, 1 is processed first, then the left sub tree, and this is followed by the right sub tree.



5.2.1 Depth First Traversal

- Pre-order traversal
  - Therefore, processing must return to the right sub tree after finishing the processing of the left sub tree.
  - To move to the right sub tree after processing the left sub tree, we must maintain the root information.

- Algorithm
  - Preorder traversal is defined as follows:
    - Visit the root.
    - Traverse the left sub tree in Preorder.
    - Traverse the right sub tree in Preorder.

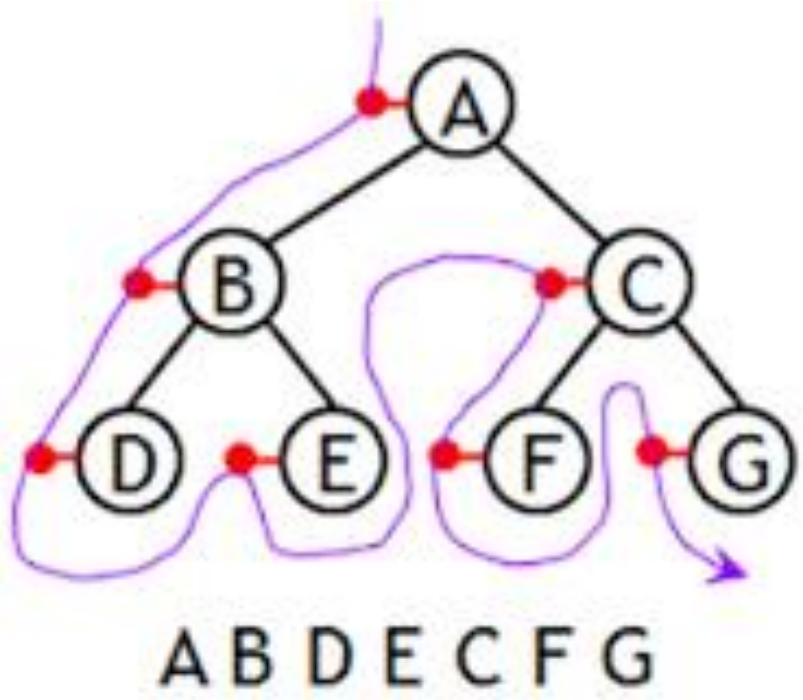
5.2.1 Depth First Traversal

- Pre-order traversal

Algorithm

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            Write TREE -> DATA
Step 3:            PREORDER(TREE -> LEFT)
Step 4:            PREORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

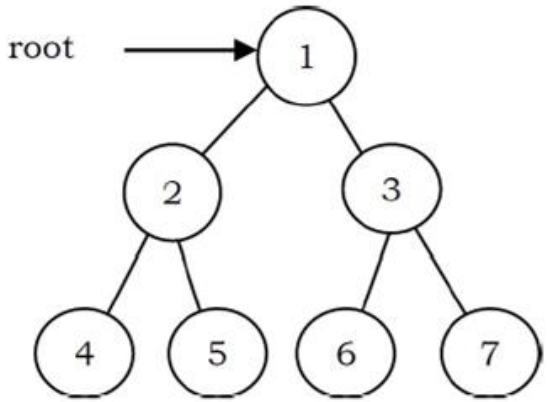
Example 1 :Traversal order



5.2.1 Depth First Traversal

- Pre-order traversal

  Example 2 :



The nodes of tree would be visited in the order: 1 2 4 5 3 6 7

5.2.1 Depth First Traversal

- Pre-order traversal

  Code segment :

```
private void preOrder(Node localRoot)
{
    if(localRoot != null)
    {
        System.out.print(localRoot.iData + " ");
        preOrder(localRoot.leftChild);
        preOrder(localRoot.rightChild);
    }
}
```

5.2.1 Depth First Traversal

- In-order traversal
  - In In-order Traversal the root is visited between the sub trees.

- Algorithm
  - In-order traversal is defined as follows:
    - Traverse the left sub tree in In-order.
    - Visit the root.
    - Traverse the right sub tree in In-order.

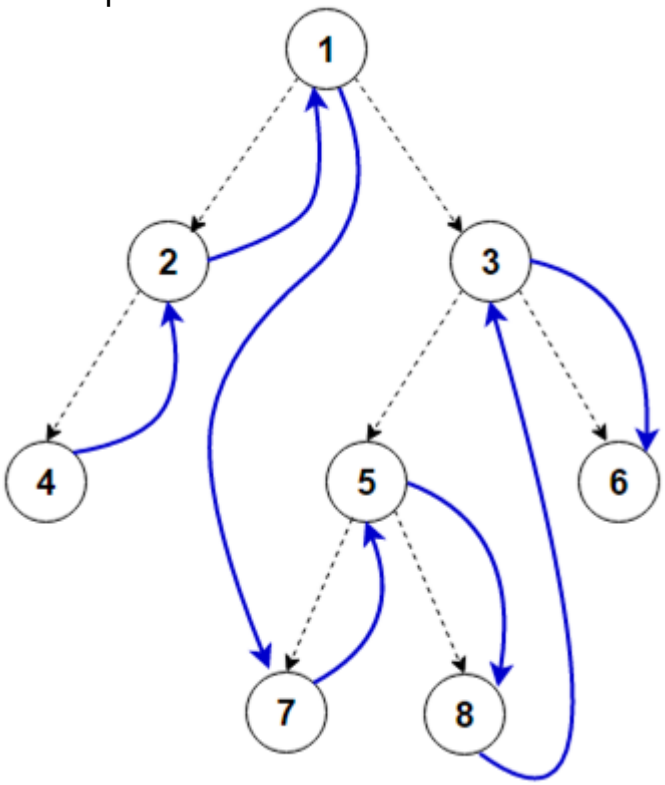
5.2.1 Depth First Traversal

- In-order traversal

Algorithm

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            INORDER(TREE -> LEFT)
Step 3:            Write TREE -> DATA
Step 4:            INORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

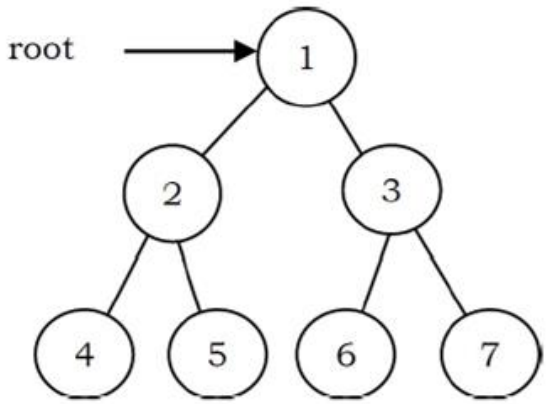
Example 1 :Traversal order



5.2.1 Depth First Traversal

- In-order traversal

  Example 2 :



The nodes of tree would be visited in the order: 4 2 5 1 6 3 7

5.2.1 Depth First Traversal

- In-order traversal

  Code segment :

```
private void inOrder(Node localRoot)
{
if(localRoot != null)
{
    inOrder(localRoot.leftChild);
    System.out.print(localRoot.iData + " ");
    inOrder(localRoot.rightChild);
}
}
```

5.2.1 Depth First Traversal

- Post-order traversal
  - In post-order traversal, the root is visited after both sub trees.

- Algorithm
  - Post-order traversal is defined as follows:
    - Traverse the left sub tree in Post-order.
    - Traverse the right sub tree in Post-order.
    - Visit the root.

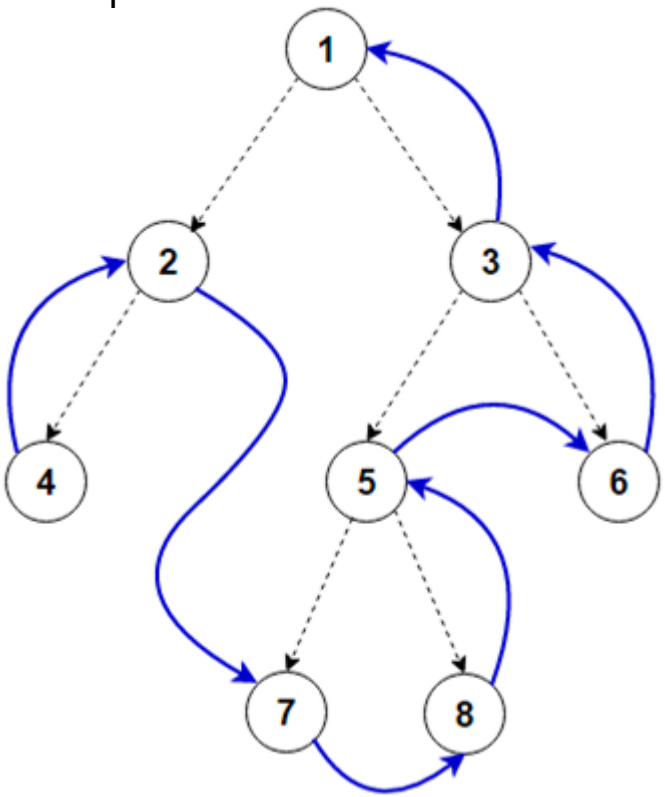
5.2.1 Depth First Traversal

- Post-order traversal

Algorithm

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            POSTORDER(TREE -> LEFT)
Step 3:            POSTORDER(TREE -> RIGHT)
Step 4:            Write TREE -> DATA
       [END OF LOOP]
Step 5: END
```

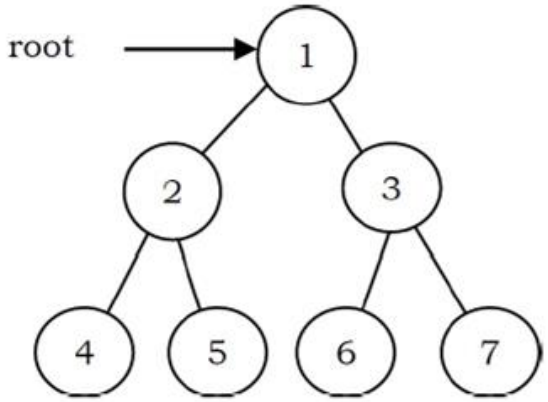
Example 1 :Traversal order



5.2.1 Depth First Traversal

- Post-order traversal

  Example 2 :



The nodes of the tree would be visited in the order: 4 5 2 6 7 3 1

5.2.1 Depth First Traversal

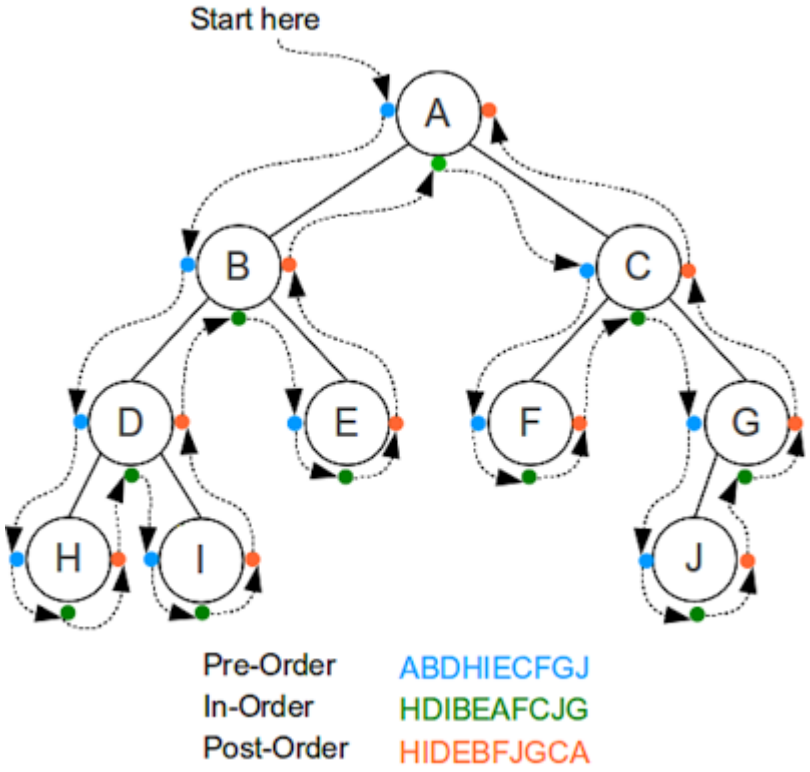
- Post-order traversal

  Code segment :

```
private void postOrder(Node localRoot)
{
    if(localRoot != null)
    {
        postOrder(localRoot.leftChild);
        postOrder(localRoot.rightChild);
        System.out.print(localRoot.iData + " ");
    }
}
```

5.2.1 Depth First Traversal

Example:



5.2.2 Breadth First Traversal

- breadth-first search (BFS) is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal.

- This algorithm is also called as the breadth-first traversal/Breadth search (BFT/BFS) Algorithm or Level order traversal algorithm

5.2.2 Breadth First Traversal

- In level-order traversal, all the nodes at a level are accessed before going to the next level.
- Level order traversal is defined as follows:
  - Visit the root.
  - While traversing level (, keep all the elements at level (+ 1 in queue.
  - Go to the next level and visit all the nodes at that level.
  - Repeat this until all levels are completed.

5.2.2 Breadth First Traversal

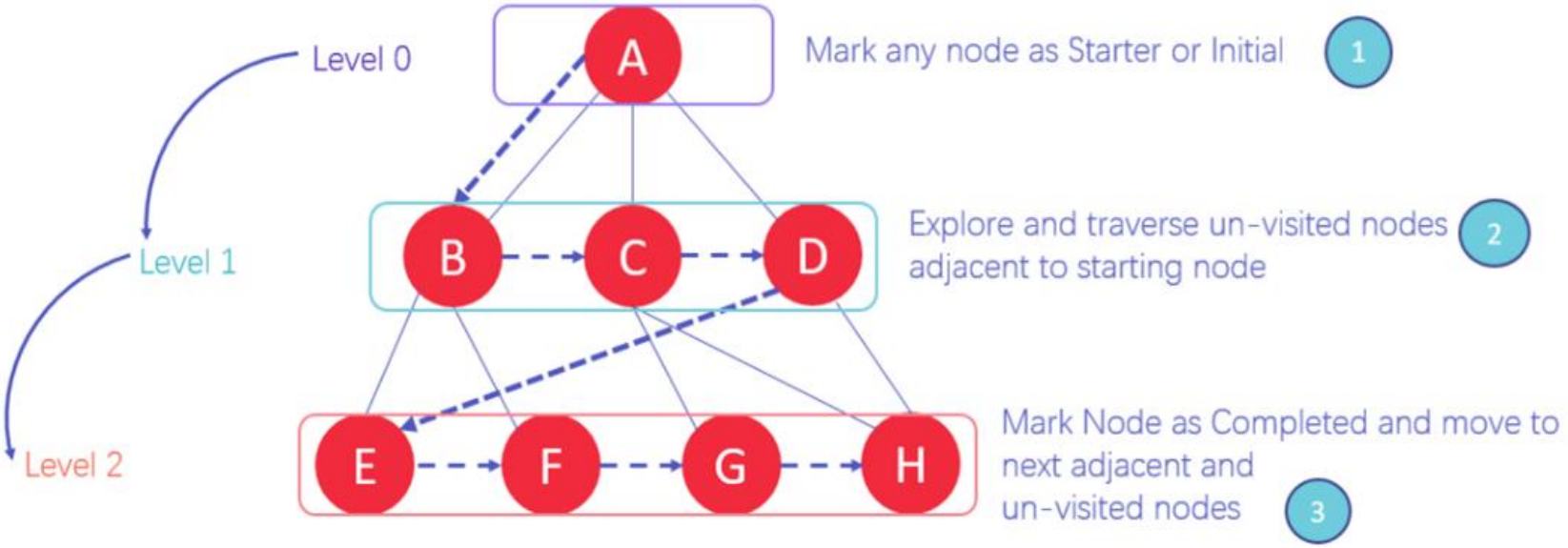
- **How it works**

  - **Breadth-first search** (BFS) is an uninformed search method that aims to expand and examine all nodes of a graph systematically in search of a solution. In other words, it exhaustively searches the entire graph without considering the goal until it finds it. It does not use a heuristic.

  - From the standpoint of the algorithm, all child nodes obtained by expanding a node are added to a FIFO queue. In typical implementations, nodes that have not yet been examined for their neighbors are placed in some container (such as a queue or linked list) called "open" and then once examined are placed in the container "closed".

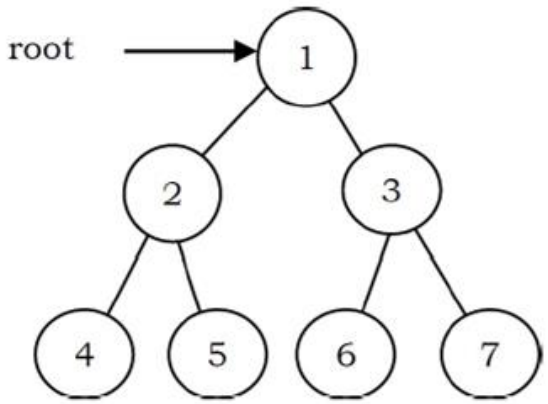
5.2.2 Breadth First Traversal

- In the breadth first traversal, the nodes are visited level by level, from left to right.



5.2.1 Breadth First Traversal

- Example:



The nodes of the tree are visited in the order: 1 2 3 4 5 6 7

5.2.2 Breadth First Traversal

- **Applications of BFS**

  - Breadth-first search can be used to solve many problems in graph theory, for example:
  - Finding all connected components in a graph.
  - Finding all nodes within one connected component
  - Copying Collection, Cheney's algorithm
  - Finding the shortest path between two nodes u and v (in an unweighted graph)
  - Testing a graph for bipartiteness
  - (Reverse) Cuthill–McKee mesh numbering

5.2.2 Breadth First Traversal

- Recursion cannot be used to implement a breadth-first traversal since the recursive calls must follow the links that lead deeper into the tree.
- The best way to save a node's children for later access is to use a queue
- We can then use an iterative loop to move across the tree in the correct node order to produce a breadth-first traversal.
- The above Listing uses a queue to implement the breadth first traversal.
- The process starts by saving the root node and in turn priming the iterative loop.
- During each iteration, we remove a node from the queue, visit it, and then add its children to the queue.
- The loop terminates after all nodes have been visited.

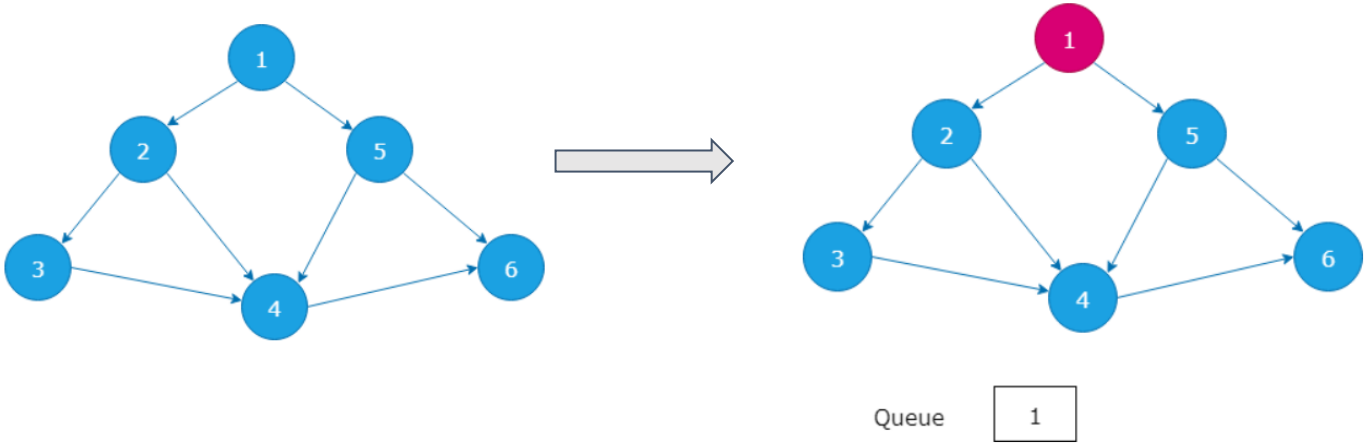
5.2.2 Breadth First Traversal

Pseudo-Code for Breadth-First Traversal

- put root node onto a queue
- while the queue is not empty
  - dequeue the next node
  - visit the node
  - enqueue the left child node
  - enqueue the right child node

5.2.2 Breadth First Traversal

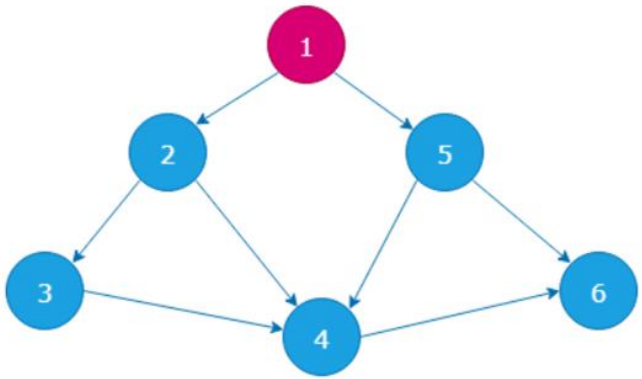
- Example

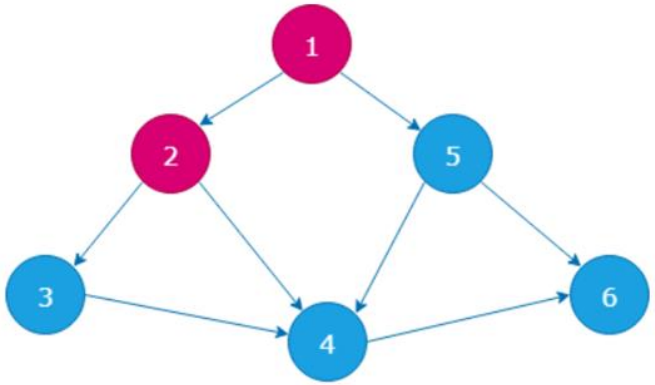


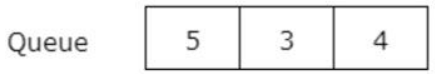
We start from node(1), and put it in the queue

5.2.2 Breadth First Traversal

- Example





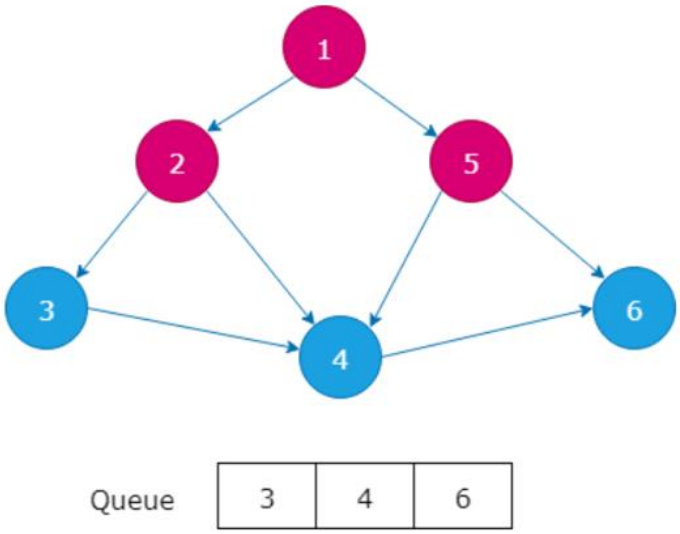


Next, we go through all the neighbors of node(1) and put all the unvisited node on the queue. node(2) and node(5) will go on to the queue and marked as visited.Traversal = {1}

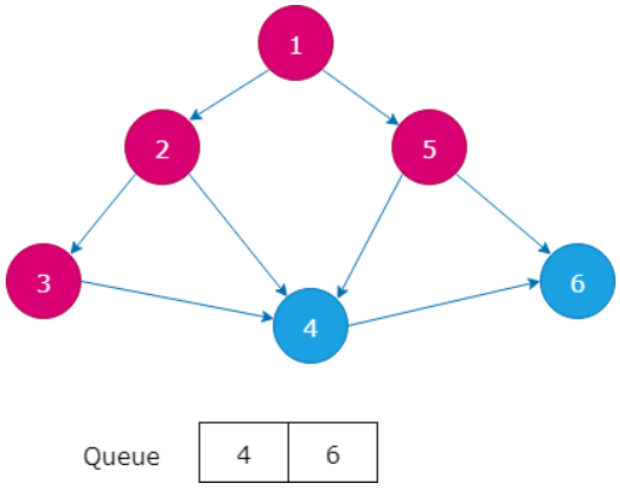
Again, we dequeue from the queue and this time we get node(2). We print it and go for all the neighbor node, node(3) and node(4) and mark them as visited.Traversal = {1,2}

5.2.2 Breadth First Traversal

- Example



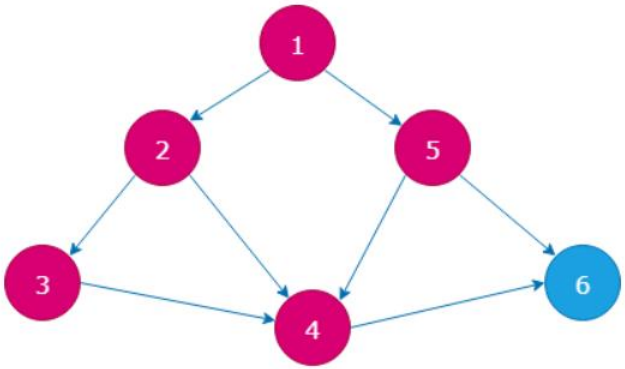
node(5) is dequeued next and printed. Here, even though node(4) is a neighbor of node(5), it is already visited and hence not put on to the queue again. But node(6) is not yet visited, so put it on to the queue.Traversal = {1,2,5}

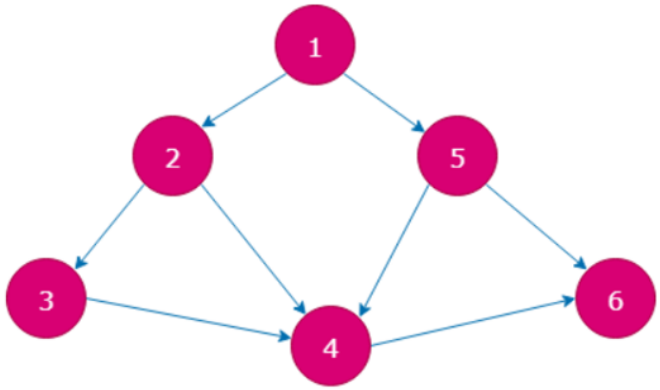


Now, we pop node(3) and print it, however, node(4) is already visited. Hence, nothing is added to the queue. Traversal = {1,2,5,3}

5.2.2 Breadth First Traversal

- Example





Queue | 6

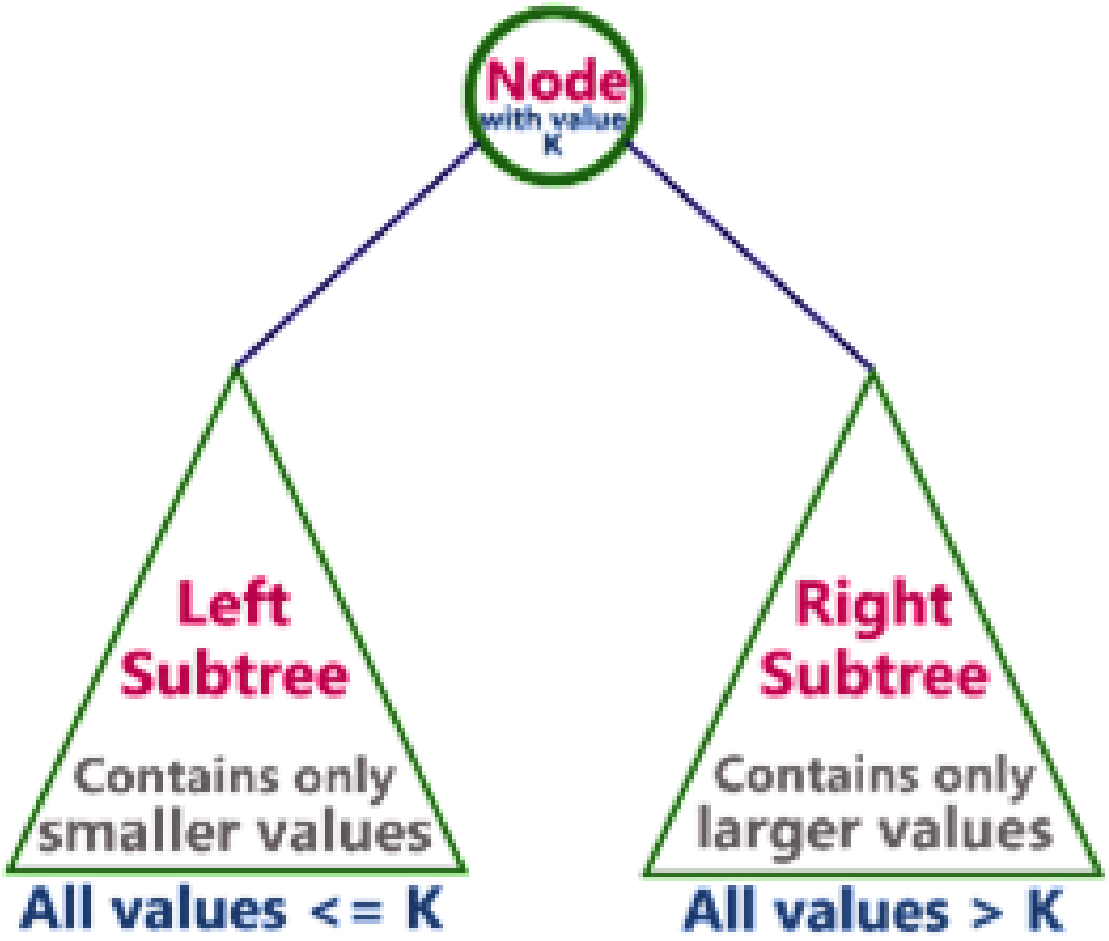
Next, node(4) is taken out from queue and printed, nothing goes on to queue. Traversal = {1,2,5,3,4}

Last, we pop node(6) and print it. Traversal = {1,2,5,3,4,6}.

5.3 Binary Search trees

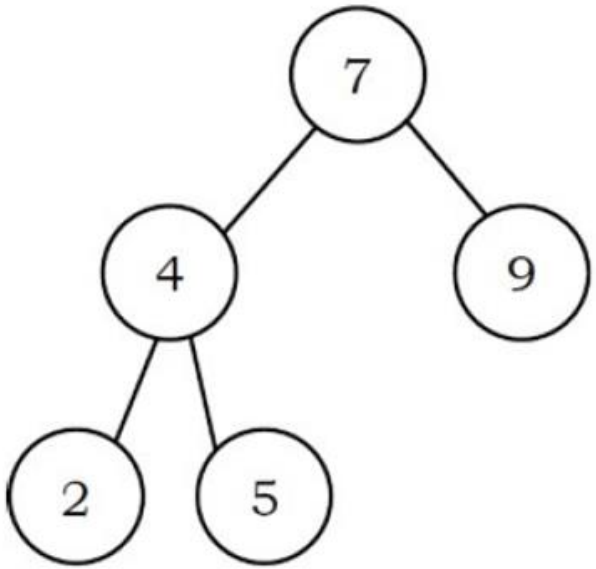
- A binary search tree, also known as an ordered binary tree, is a variant of binary trees in which the nodes are arranged in an order.
- In binary search trees, all the left sub tree elements should be less than root data and all the right sub tree elements should be greater than root data. This is called binary search tree property.
- In a binary search tree, each non leaf node v stores an element e such that
  - the elements stored in the left sub tree of v are less than e.
  - the elements stored in the right sub tree of v are greater than or equal to e.
- Both the left and right sub trees must also be binary search trees.

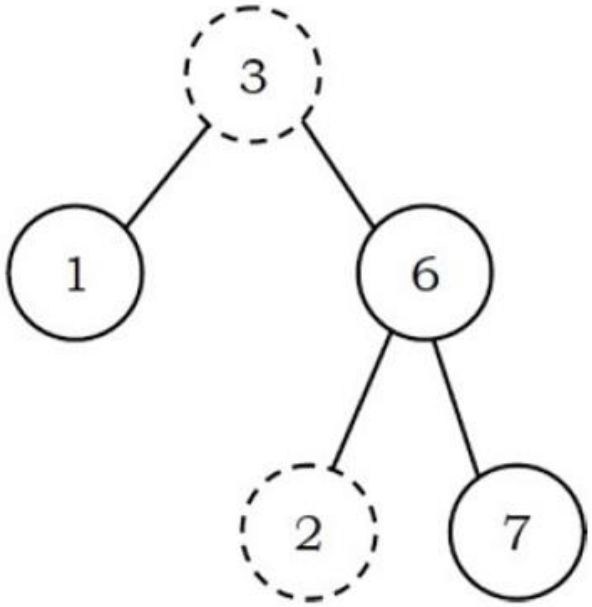
5.3 Binary Search trees



5.3 Binary Search trees

Example: The left tree is a binary search tree and the right tree is not a binary search tree (at node 6 it's not satisfying the binary search tree property).





5.3 Binary Search trees

- **Operations on Binary Search Trees**
  - Searching an element in a binary search tree
  - Find the minimum element from a binary search tree
  - Find the maximum element from a binary search tree
  - Inserting an element in a binary search tree
  - Deleting an element from a binary search tree

5.3 Binary Search trees

- **Finding an Element in Binary Search Trees**
  - Find operation is straightforward in a BST.
  - Start with the root and keep moving left or right using the BST property.
  - If the data we are searching is same as nodes data then we return current node.
  - If the data we are searching is less than nodes data then search left sub tree of current node; otherwise search right sub tree of current node.
  - If the data is not present, we end up in a NULL link.

5.3 Binary Search trees

- **Finding an Element in Binary Search Trees**

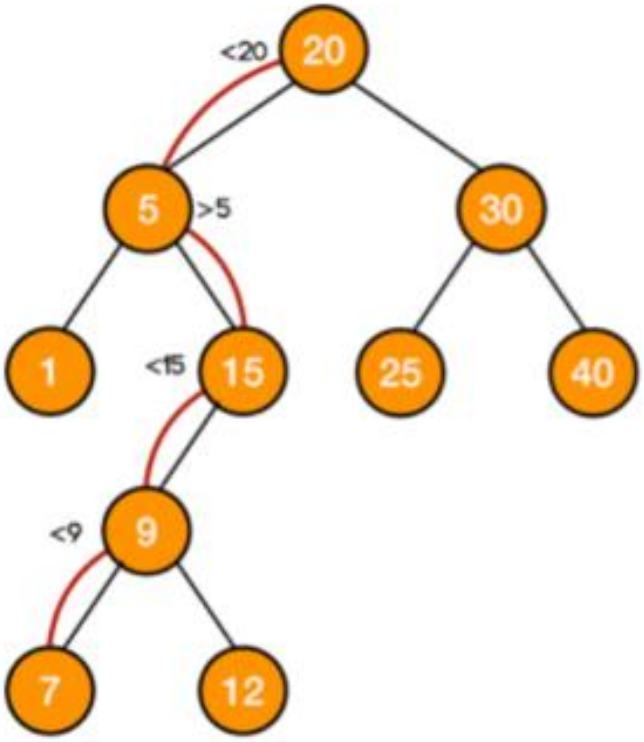
  Algorithm:

```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
           Return searchElement(TREE -> LEFT, VAL)
         ELSE
           Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

5.3 Binary Search trees

- **Finding an Element in Binary Search Trees**

  Example 1: Find element 7

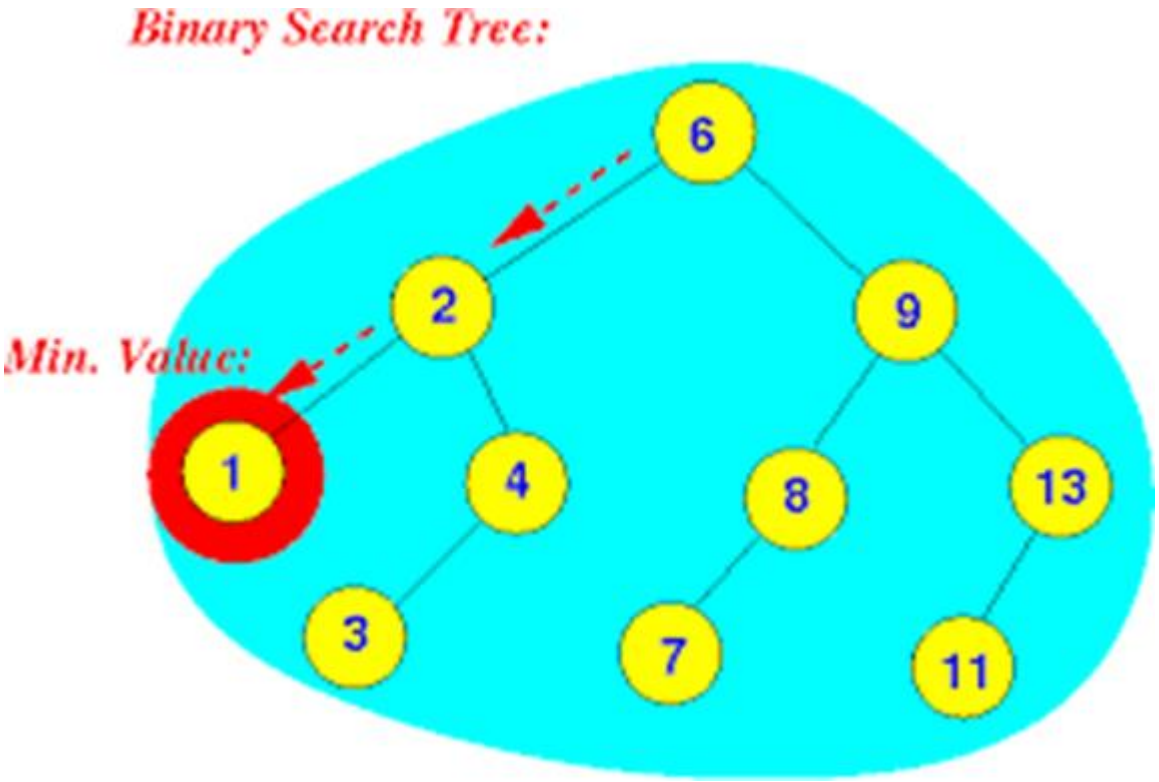


SEARCH FOR 7
STEP 1: 7 IS SMALLER THAN 20: GO LEFT
STEP 2: 7 IS GREATER THAN 5: GO RIGHT
STEP 3: 7 IS SMALLER THAN 15: GO LEFT
STEP 4: 7 IS SMALLER THAN 9: GO LEFT

5.3 Binary Search trees

- **Finding the minimum element from a Binary Search Trees**

  Example 1:



5.3 Binary Search trees

- **Find the minimum element from a binary search tree**

  Algorithm:

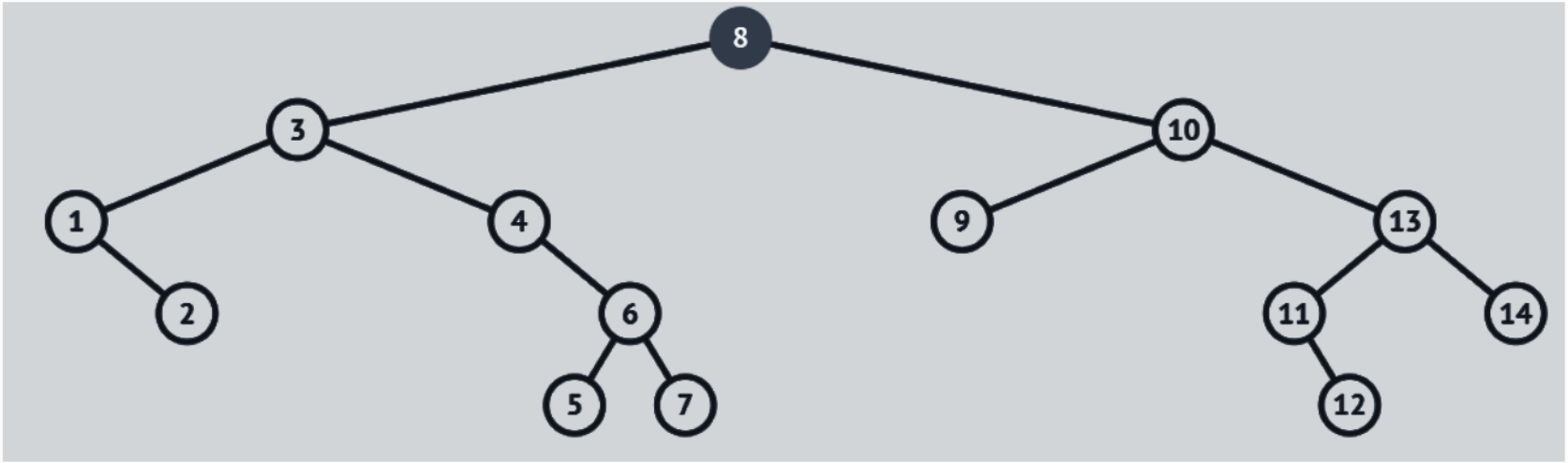
```
findSmallestElement(TREE)

Step 1: IF TREE = NULL OR TREE -> LEFT = NULL
            Returen TREE
        ELSE
            Return findSmallestElement(TREE -> LEFT)
        [END OF IF]
Step 2: END
```

5.3 Binary Search trees

- **Find the minimum element from a binary search tree**

  Example 2:

start from root i.e 8.

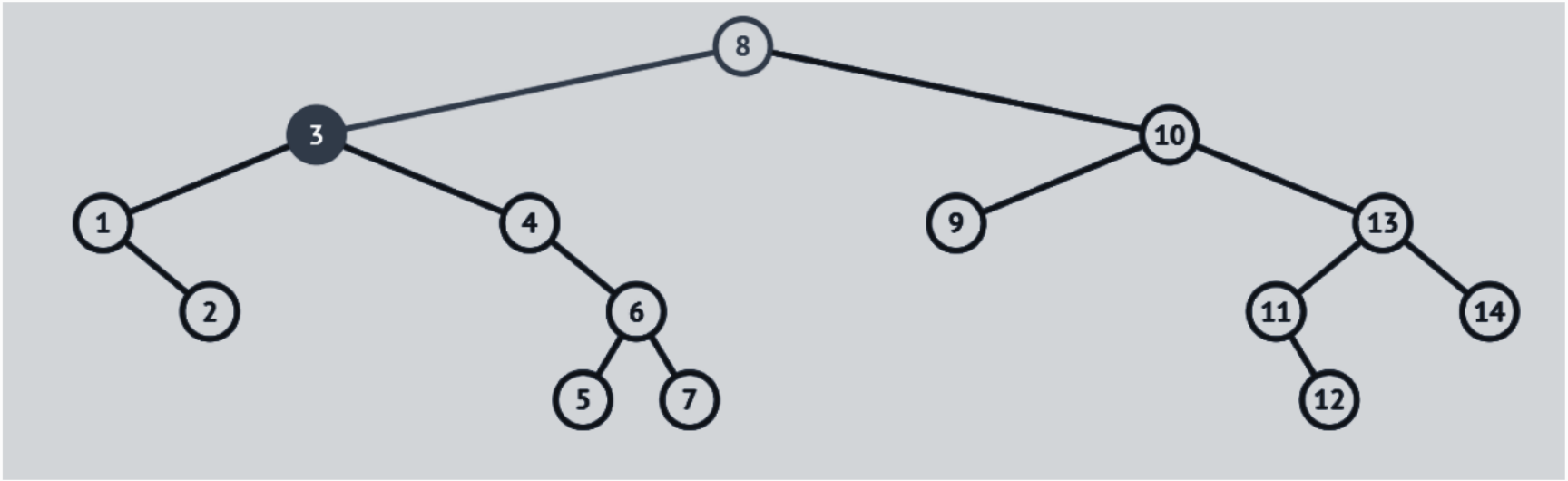


5.3 Binary Search trees

- **Find the minimum element from a binary search tree**

  Example 2:

As left of root is not null go to left of root i.e 3.



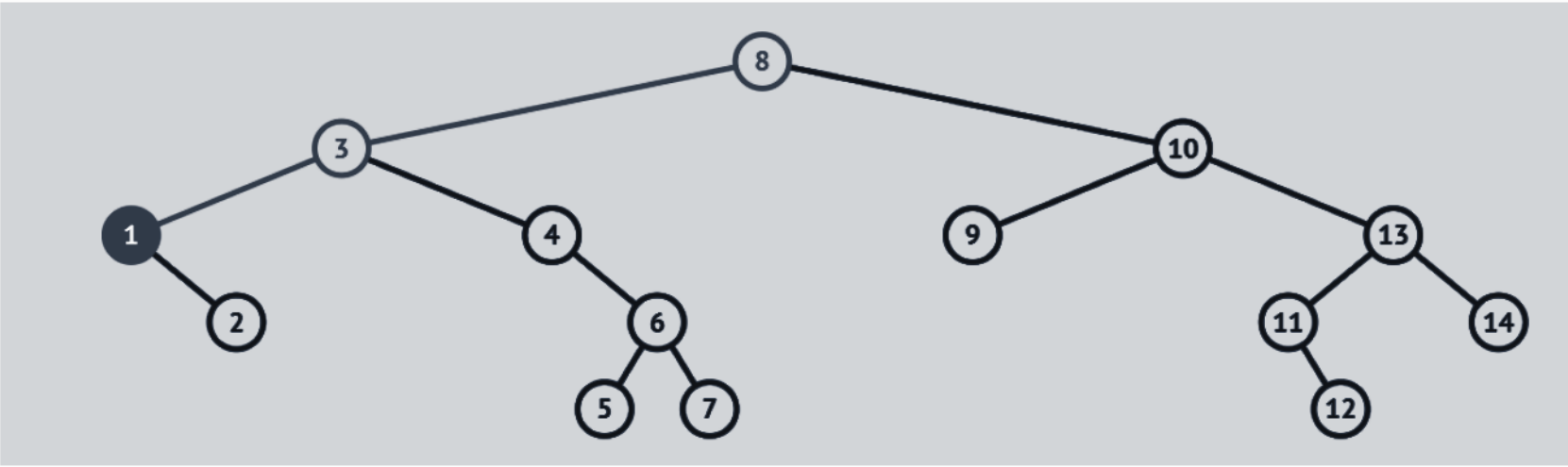
5.3 Binary Search trees

- **Find the minimum element from a binary search tree**

  Example 2:

As left of 3 is not null go to left of 3 i.e. 1.

Now as the left of 1 is null therefore 1 is the minimum element



5.3 Binary Search trees

- **Find the maximum element from a binary search tree**

  Algorithm:

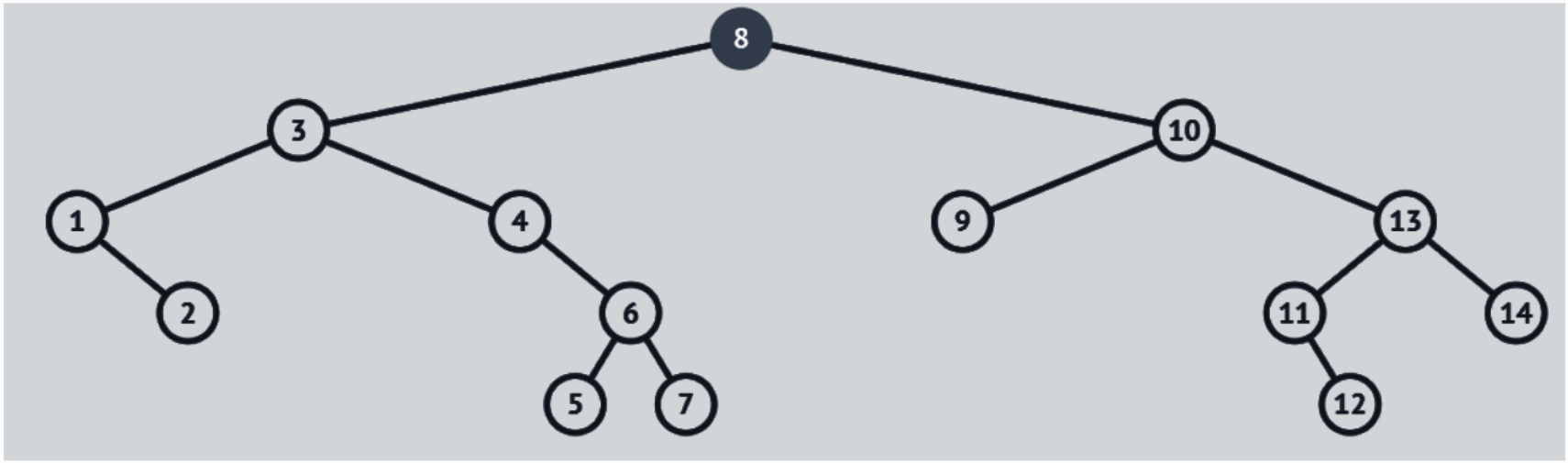
```
findLargestElement(TREE)

Step 1: IF TREE = NULL OR TREE -> RIGHT = NULL
            Return TREE
        ELSE
           Return findLargestElement(TREE -> RIGHT)
        [END OF IF]
Step 2: END
```

5.3 Binary Search trees

- **Find the maximum element from a binary search tree**

  Example:

start from root i.e 8.

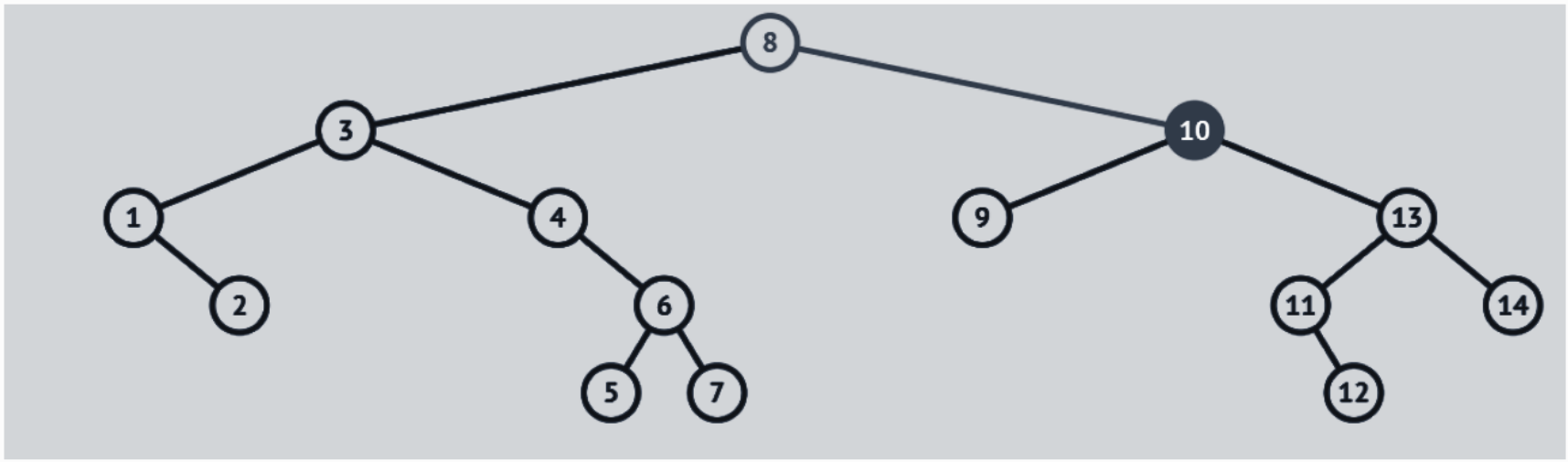


5.3 Binary Search trees

- **Find the maximum element from a binary search tree**

  Example:

As right of root is not null go to right of root i.e 10.

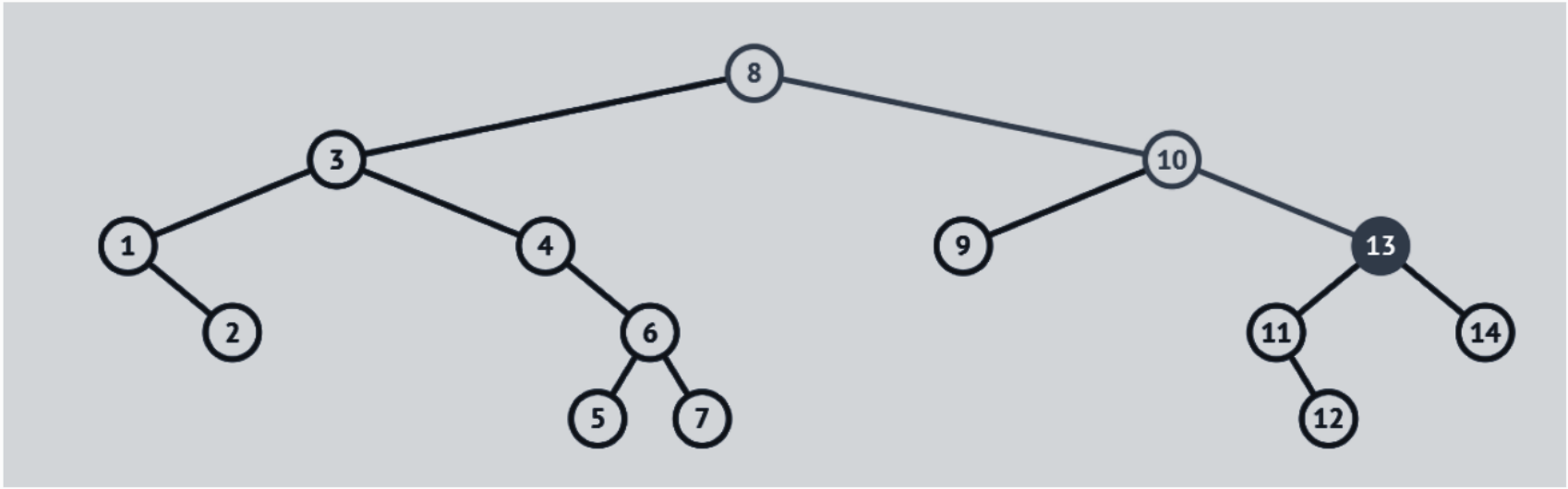


5.3 Binary Search trees

- **Find the maximum element from a binary search tree**

  Example:

As right of 10 is not null go to right of root i.e 13.



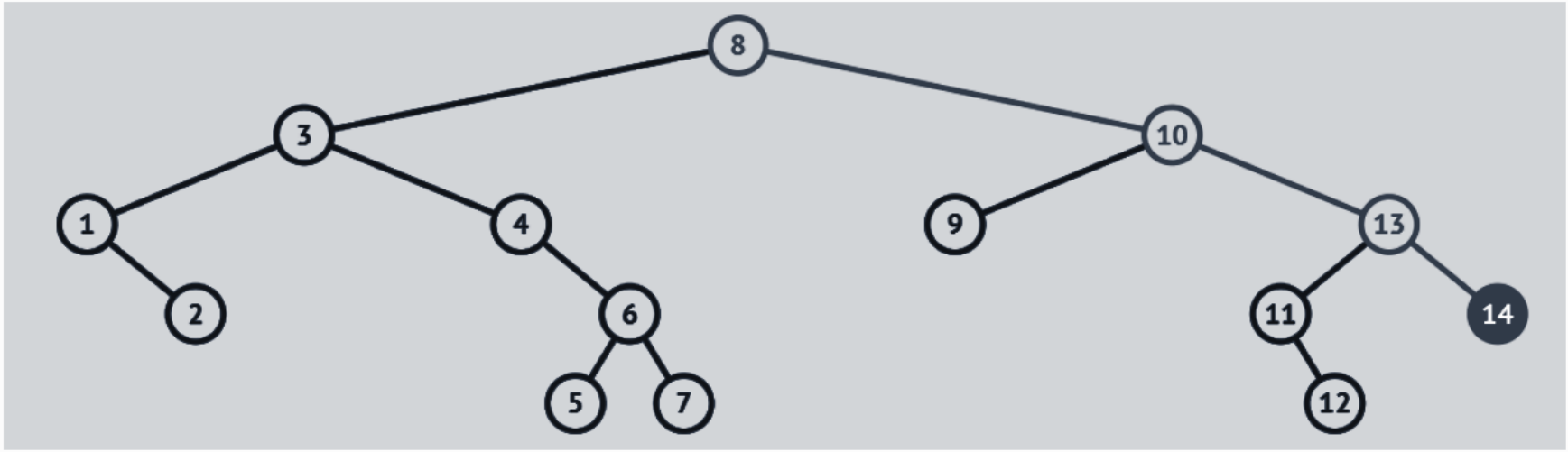
5.3 Binary Search trees

- **Find the maximum element from a binary search tree**

  Example:

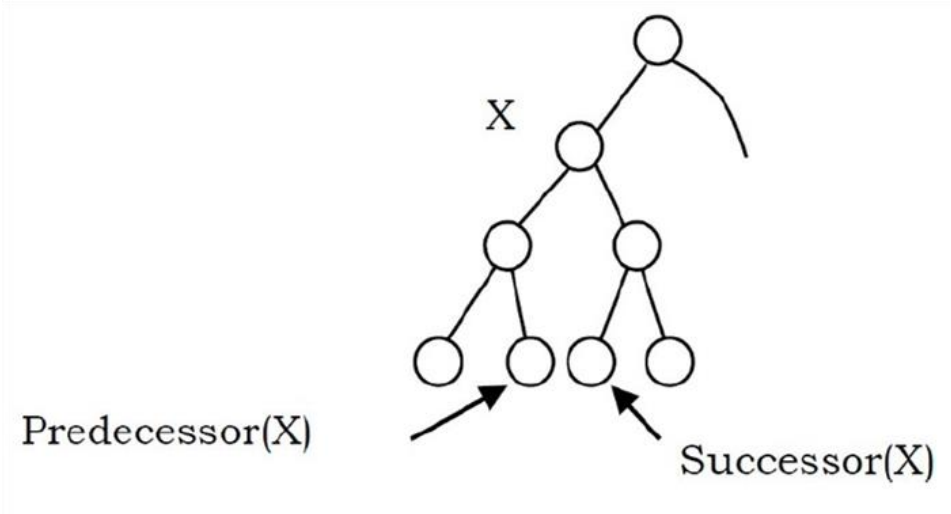
As right of 13 is not null go to right of root i.e 14.

Now as the right of 14 is null therefore 14 is the maximum element.



5.3 Binary Search trees

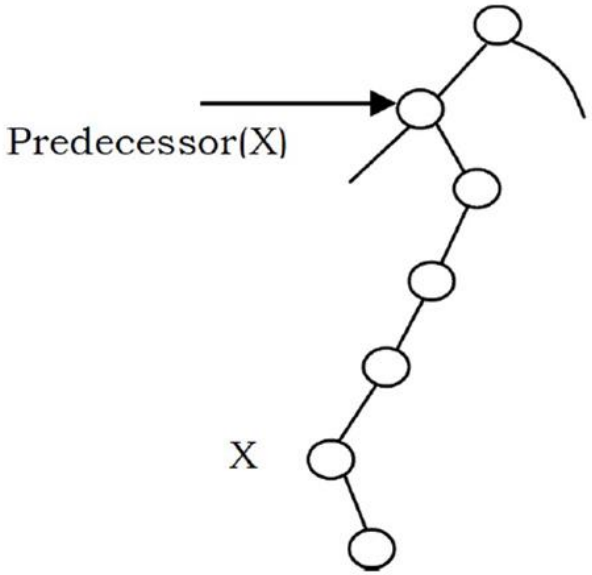
- **Inorder Predecessor and Successor**
  - If X has two children then its inorder predecessor is the maximum value in its left sub tree and its inorder successor the minimum value in its right sub tree.



5.3 Binary Search trees

- **Inorder Predecessor and Successor**

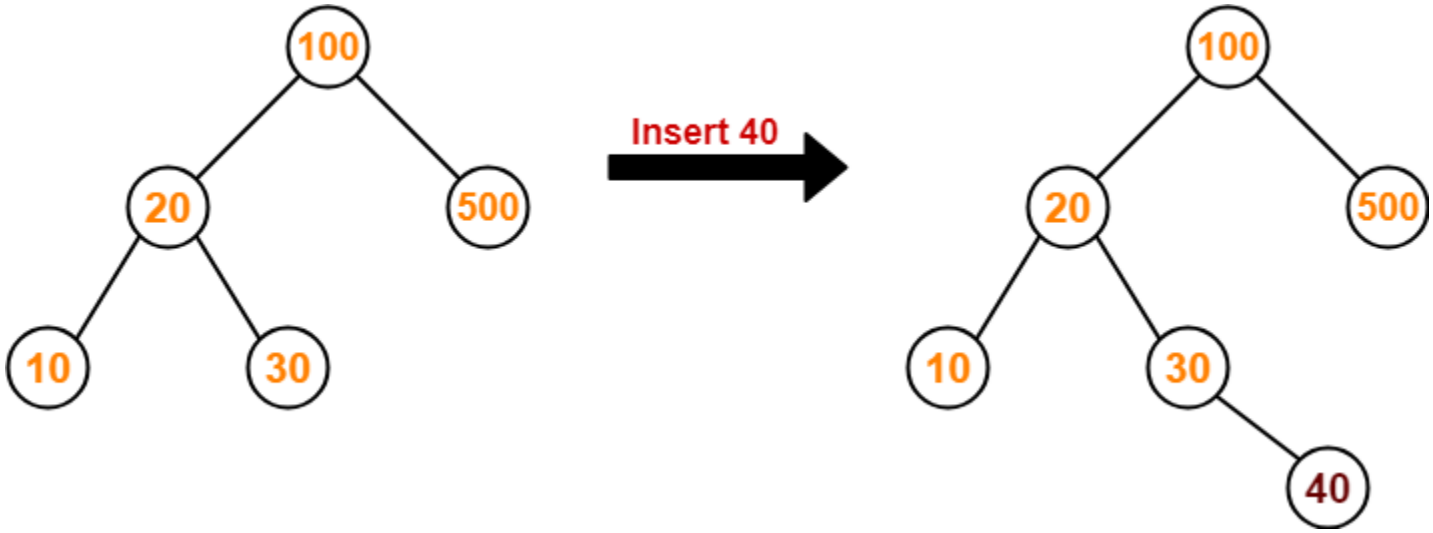
  If it does not have a left child, then a node's inorder predecessor is its first left ancestor



5.3 Binary Search trees

- **Inserting an Element to Binary Search Tree**

  Example 1:



5.3 Binary Search trees

- **Inserting an Element to Binary Search Tree**
  - To insert data into binary search tree, first we need to find the location for that element.
  - We can find the location of insertion by following the same mechanism as that of find operation.
  - While finding the location, if the data is already there then we can simply neglect and come out.
  - Otherwise, insert data at the last location on the path traversed.

5.3 Binary Search trees

- **Inserting an Element to Binary Search Tree**

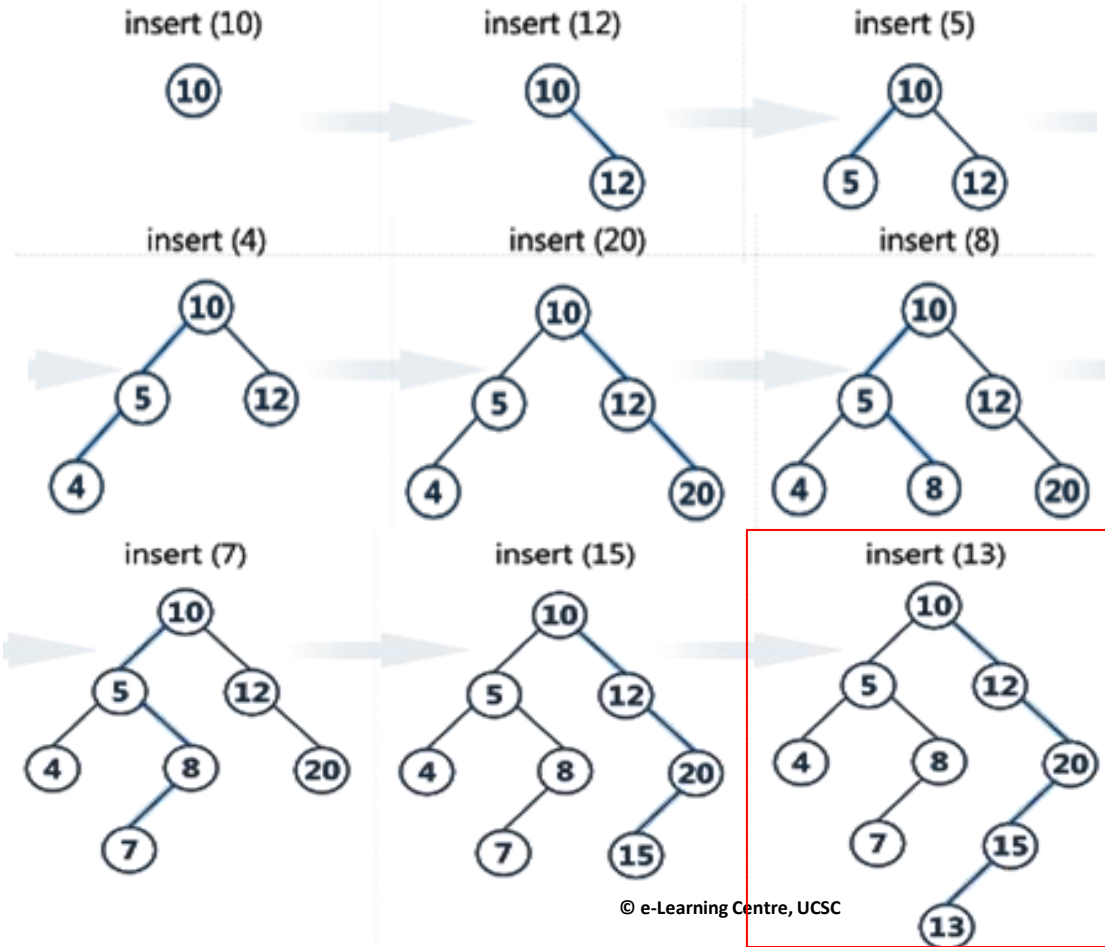
  Algorithm:

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

5.3 Binary Search trees

- **Inserting an Element from Binary Search Tree**

  Example : Insert 10,12,5,4,20,8,7,15,13 to a Binary Search Tree



5.3 Binary Search trees

- **Inserting an Element from Binary Search Tree**

  Java implementation

```
public void insert(int id, double dd){
    Node newNode = new Node(); // make new node
    newNode.iData = id; // insert data
    newNode.dData = dd;
    if(root==null) // no node in root
        root = newNode;
    else // root occupied
    {
    Node current = root; // start at root
    Node parent;
    while(true) // (exits internally)
    {
        parent = current;
        if(id < current.iData) // go left?
{
```

5.3 Binary Search trees

- **Inserting an Element from Binary Search Tree**

  Java implementation

```
            current = current.leftChild;
            if(current == null) // if end of the line,
            { // insert on left
            parent.leftChild = newNode;
            return;
            }
        } // end if go left
```

5.3 Binary Search trees

- **Inserting an Element from Binary Search Tree**

  Java implementation

```
else // or go right?
{
    current = current.rightChild;
    if(current == null) // if end of the line
        { // insert on right
        parent.rightChild = newNode;
        return;
        }
    } // end else go right
    } // end while
} // end else not root
} // end insert()
```

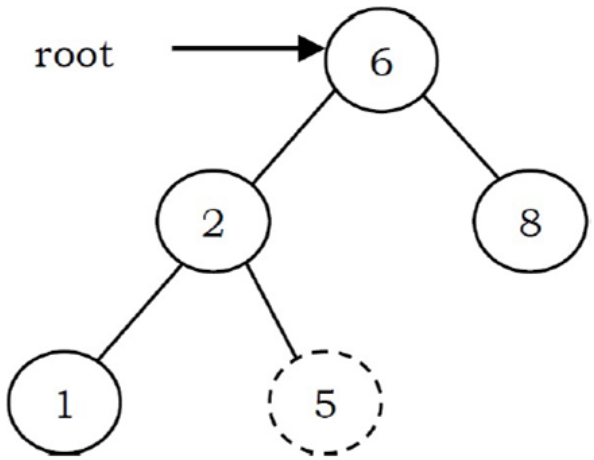
5.3 Binary Search trees

- **Deleting an Element from Binary Search Tree**
  - The delete operation is more complicated than other operations.
  - This is because the element to be deleted may not be the leaf node.
  - In this operation also, first we need to find the location of the element which we want to delete.
  - Once we have found the node to be deleted, consider the following cases:

5.3 Binary Search trees

- Case 1: The Node to Be Deleted Has No Children
  - If the element to be deleted is a leaf node: return NULL to its parent. That means make the corresponding child pointer NULL
  - In the tree below to delete 5, set NULL to its parent node 2.

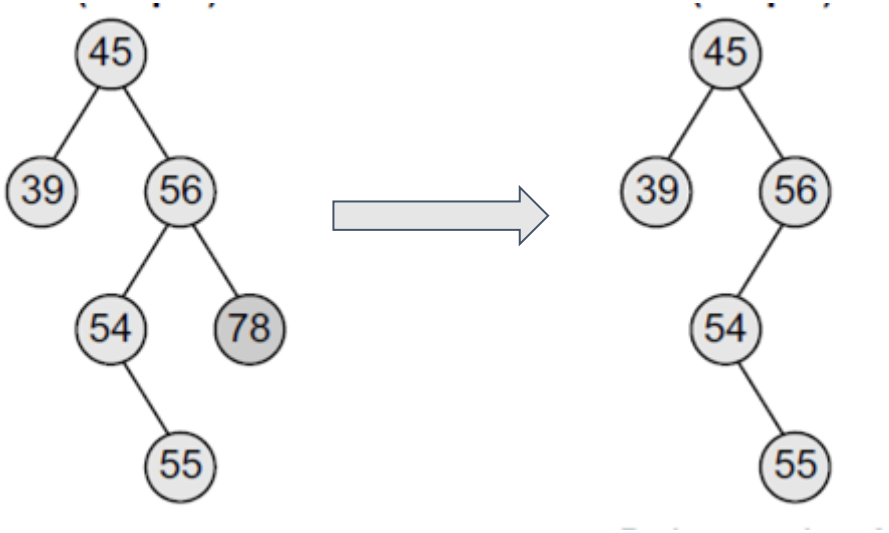
Example :



5.3 Binary Search trees

- Case 1: The Node to Be Deleted Has No Children

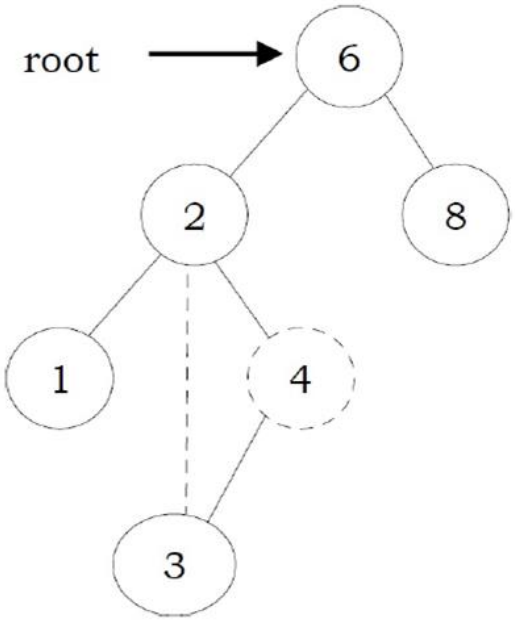
  Example 2 : Delete 78



5.3 Binary Search trees

- Case 2: The Node to Be Deleted Has One Child
  - If the element to be deleted has one child: In this case we just need to send the current node's child to its parent.
  - In the tree below, to delete 4, 4 left sub tree is set to its parent node 2.

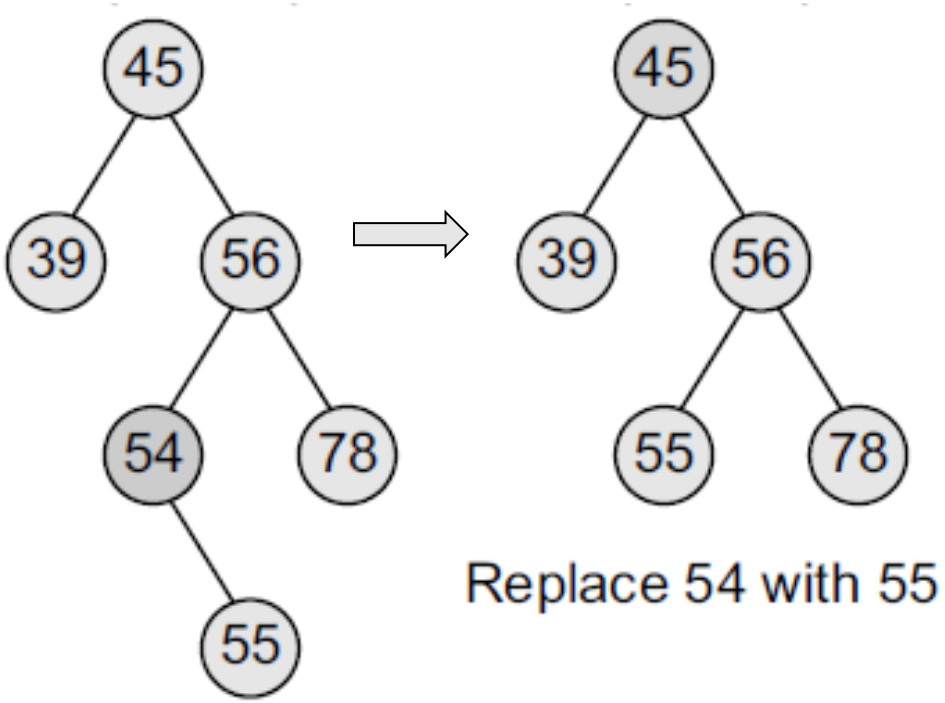
    Example :



5.3 Binary Search trees

- Case 2: The Node to Be Deleted Has One Child

  Example 2: Delete 54



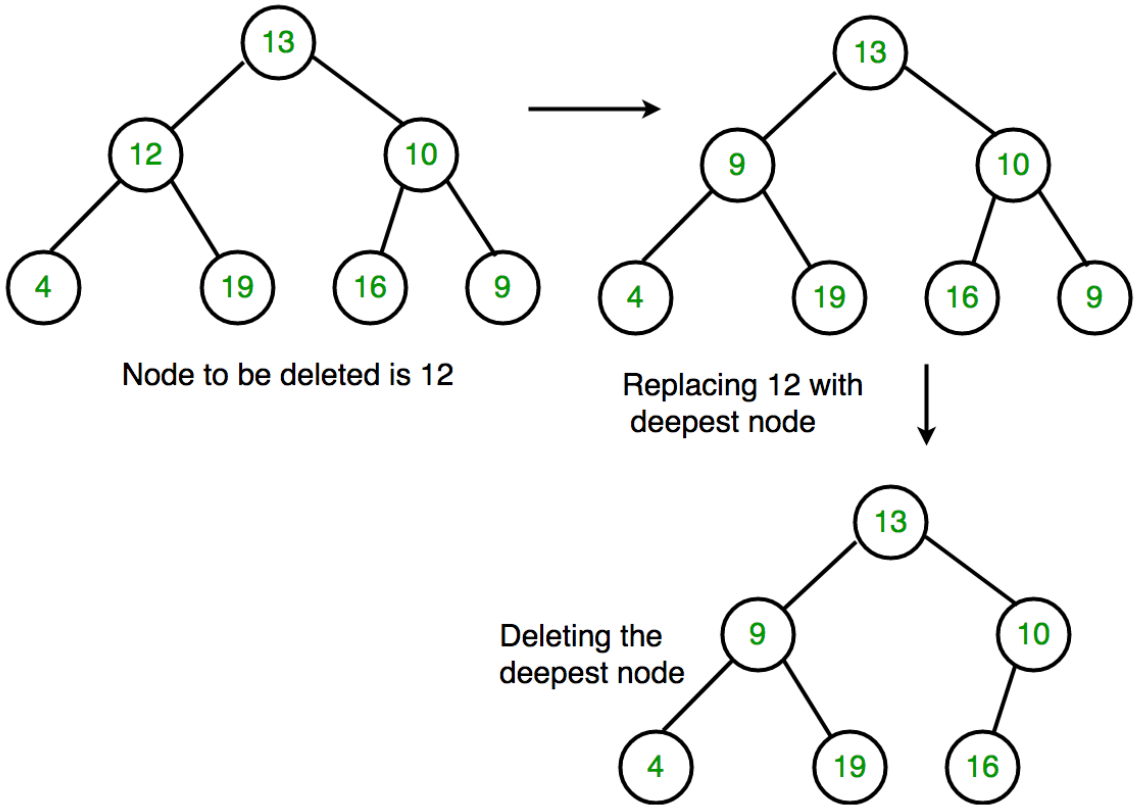
5.3 Binary Search trees

- Case 3: The Node to Be Deleted Has Two Children
  - If the deleted node has two children, you can't just replace it with one of these children, at least if the child has its own children.
  - this method we shall use is to replace the node being deleted by the rightmost node in its left sub-tree Or Leftmost mode in its right-sub-tree.

5.3 Binary Search trees

- Case 3: The Node to Be Deleted Has Two Children

  Example:



5.3 Binary Search trees

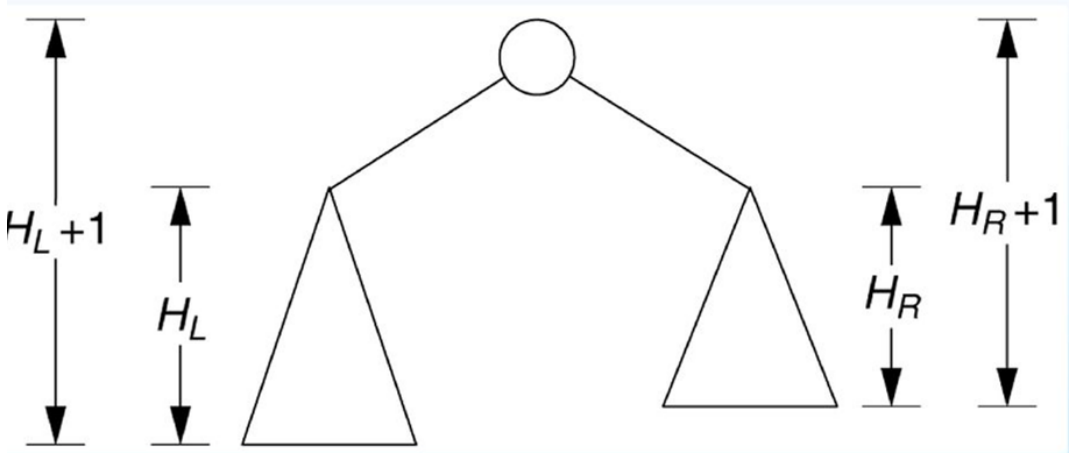
- Deleting an Element from Binary Search Tree

Algorithm (All three cases):

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

5.3 Binary Search trees

- **Height of a Binary Search Tree**
  - In order to determine the height of a binary search tree, we calculate the height of the left sub-tree and the right sub-tree.
  - Recursive view used to calculate the height of a tree : $H_T = \max(H_L + 1, H_R + 1)$



5.3 Binary Search trees

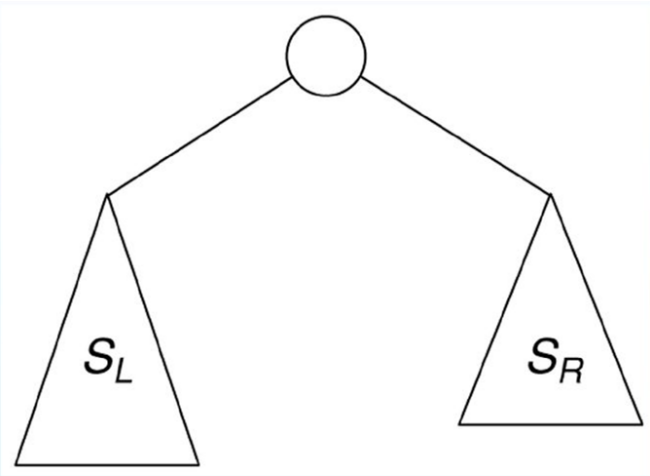
- **//* Return the height of the binary tree rooted at t.**

```
If (t = = null)
      return 0;
else
      return 1+maximum(height(t->left),height(t- >right);
```

5.3 Binary Search trees

- **Size of a Binary Search tree**

$$S_T = S_L + S_R + 1$$



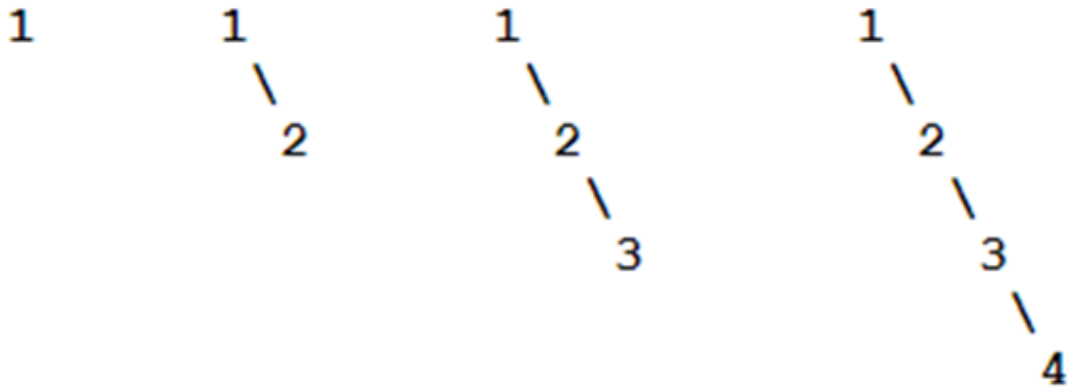

```
//* Return the size of the binary tree
rooted at t.
if (t = = null)
      return 0
else
return 1+size(t->left)+ size(t->right)
```

5.3 Binary Search trees

- **Balanced Binary Search Trees**
  - Binary search trees allow binary search for fast lookup, insertion and deletion of data items, and can be used to implement dynamic sets and lookup tables.
  - The order of nodes in a BST means that each comparison skips about half of the remaining tree, so the whole lookup takes time proportional to the binary logarithm of the number of items stored in the tree.
  - Binary search trees are a nice idea, but they fail to accomplish our goal of doing lookup, insertion and deletion each in time O(log2(n)), when there are n items in the tree. Imagine starting with an empty tree and inserting 1, 2, 3 and 4, in that order

5.3 Binary Search trees

- **Balanced Binary Search Trees**
  - Binary search trees are a nice idea, but they fail to accomplish our goal of doing lookup, insertion and deletion each in time O(log2(n)), when there are n items in the tree.
  - Imagine starting with an empty tree and inserting 1, 2, 3 and 4, in that order.



5.3 Binary Search trees

- **Self-Balancing Binary Search Trees**
  - A self-balancing binary search tree (BST) is a binary search tree that automatically tries to keep its height as minimal as possible at all times (even after performing operations such as insertions or deletions).
  - Hence having the height as small as possible is better when it comes to performing a large number of operations. Hence, self-balancing BSTs were introduced which automatically maintain the height at a minimum.
  - However, you may think having to self-balance every time an operation is performed is not efficient, but this is compensated by ensuring a large number of fast operations which will be performed later on the BST.

5.3 Binary Search trees

- **Self-Balancing Binary Search Trees**

A binary tree with height h can have at most $2^0+2^1+..2^h = 2^{(h+1)} -1$ nodes

$$n \leq 2^{(h+1)} - 1$$

$$h \geq [\log_2(n+1) - 1] \geq [\log_2(n)]$$

Hence, for self-balancing BSTs, the minimum height must always be $\log_2(n)$ rounded down. Moreover, a binary tree is said to be balanced if the height of left and right children of every node differ by either -1, 0 or +1. This value

> *Balance factor = Height of the left subtree - Height of the right subtree*

5.3 Binary Search trees

- **How do Self-Balancing Binary Search Trees Balance?**

When it comes to self-balancing, BSTs perform **rotations** after performing insert and delete operations.

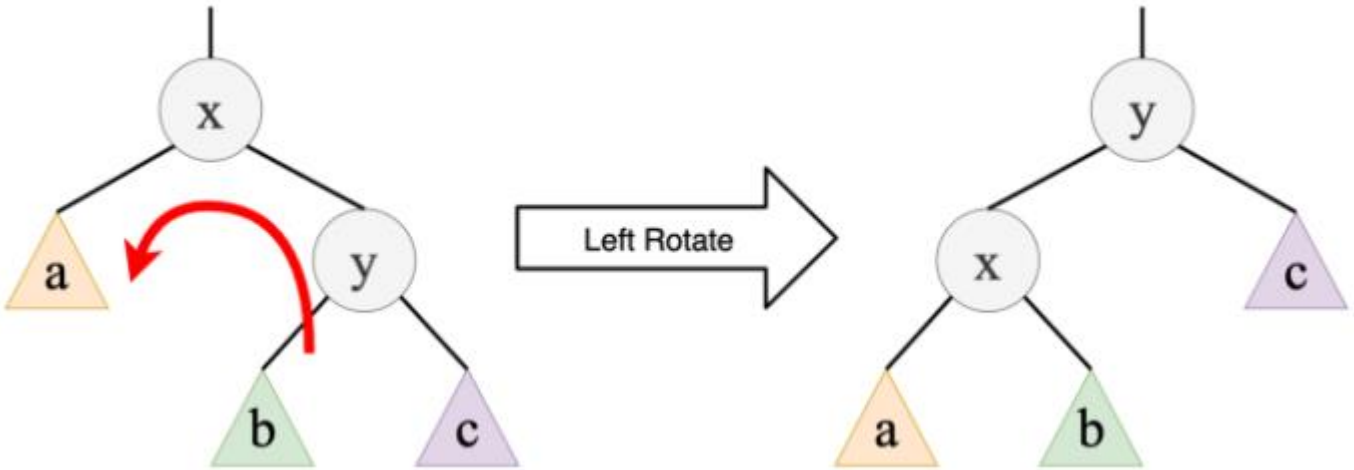
Given below are the two types of rotation operations that can be performed to balance BSTs without violating the binary-search-tree property.

- Left rotation
- Right rotation

5.3 Binary Search trees

- **Left rotation**

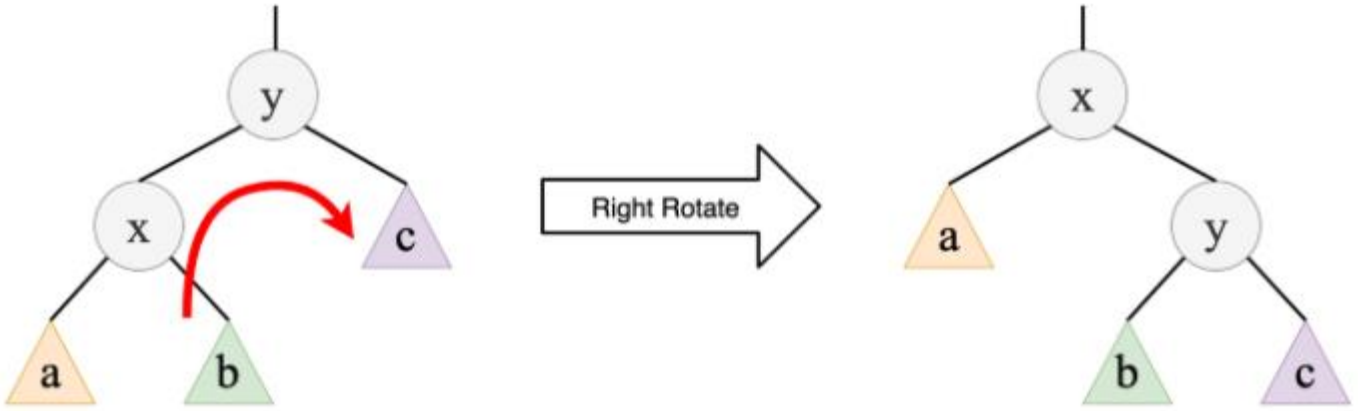
When we left rotate about node x, node y becomes the new root of the subtree. Node x becomes the left child of node y and subtree b becomes the right child of node x.



5.3 Binary Search trees

- **Right rotation**

When we right rotate about node y, node x becomes the new root of the subtree. Node y becomes the right child of node x and subtree b becomes the left child of node y.



5.3 Binary Search trees

- **Few types of BSTs that are self-balancing.**

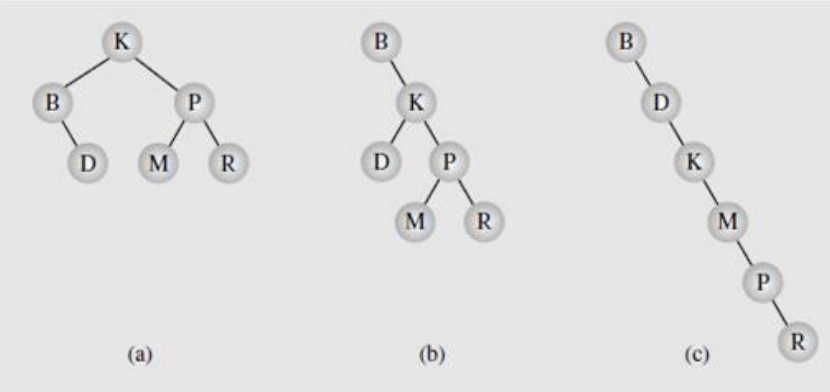
  - AVL trees
  - Red-black trees
  - Splay trees
  - Treaps

5.4 Tree Balancing

- BSTs where introduced because in theory they give nice fast search time.
- We have seen that depending on how the data arrives the tree can degrade into a linked list
- So what is a good programmer to do.
- Of course, they are to balance the tree

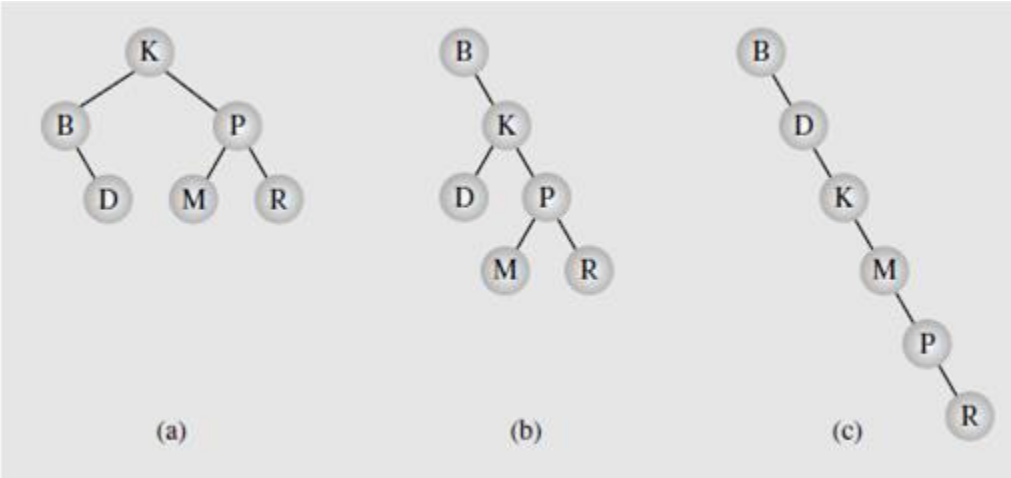
5.4.1 Introduction to tree balancing

Consider the following three examples



- A binary tree is height-balanced or simply balanced if the difference in height of both sub trees of any node in the tree is either zero or one.
- For example, for node K in Figure b, the difference between the heights of its sub trees being equal to one is acceptable. But for node B this difference is three, which means that the entire tree is unbalanced.

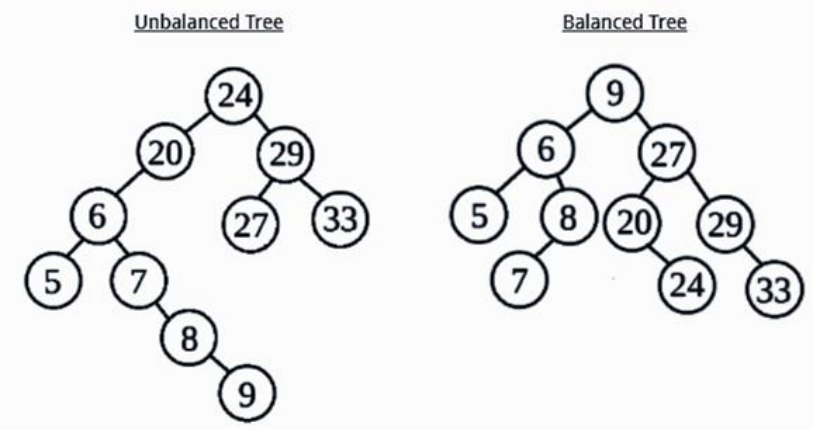
5.4.1 Introduction to tree balancing



- For the same node B in c, the difference is the worst possible, namely, five. Also, a tree is considered perfectly balanced if it is balanced and all leaves are to be found on one level or two levels.

5.4.1 Introduction to tree balancing

- There are a number of techniques to properly balance a binary tree.
- Some of them consist of constantly restructuring the tree when elements arrive and lead to an unbalanced tree.
- Some of them consist of reordering the data themselves and then building a tree, if an ordering of the data guarantees that the resulting tree is balanced.



5.4.1 Introduction to tree balancing

- We have seen the searching benefits of a balance tree, but how to we have a balance a tree
- Consider a tree with 10,000 nodes
- At its least efficient (linked list) at most 10,000 elements need to be tested find the element
- When balance, just 14 tests need to be performed
- The height of the tree is lg(10,000)=12,289=14
- Assume all the data is in a stored array.
  - The middle element becomes the root
  - The middle element of one half becomes a child
  - The idle element of the other half becomes the other child.
  - Then recursive…..

5.4.1 Introduction to tree balancing

- **How to balance?**

```
Template < T>
Void BST <T>:: Balance (T Data[], int First, int last){
If (first<=last) {
   int middle=(first+last)/2;
   insert(data[middle]);
   balance(data, first,middle-1);
   balance (data,middle+1, last);
    }
}
```

- This algorithm is quite inefficient as it relies on using extra space to store an array of values, and all values must be in this array(perhaps by in-order traversal).
- Let look at some alternative.

5.4.2 Global tree balancing (The DSW algorithm)

- The very elegant DSW algorithm was devised by Colin Day and later improved by Quentin F. Stout and Bette L.Warren.
- The building block for tree transformations in this algorithm is the rotation.
- There are two types of rotation, left and right, which are symmetrical to one another.
- The right rotation of the node Ch about its parent Par is performed according to the following algorithm:

```
rotateRight (Gr, Par, Ch)
  if Par is not the root of the tree // i.e., if Gr is not null
    grandparent Gr of child Ch becomes Ch's parent;
  right subtree of Ch becomes left subtree of Ch's parent Par;
  node Ch acquires Par as its right child;
```

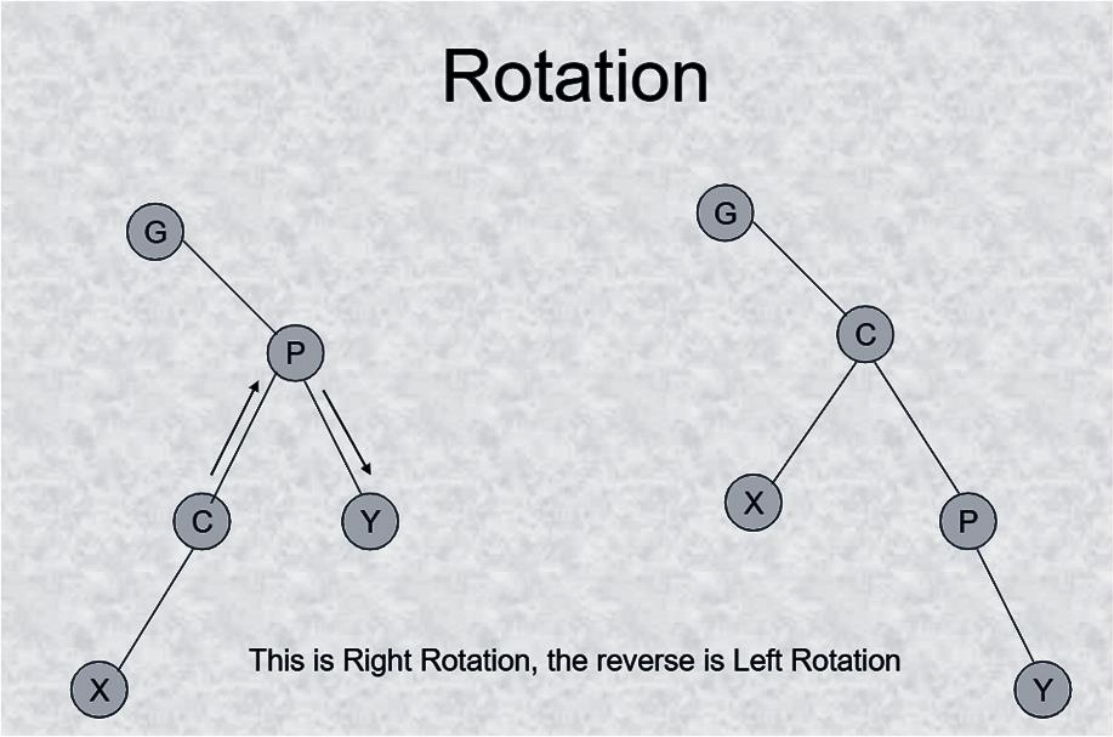
5.4.2 Global tree balancing (The DSW algorithm)

The steps involved in this compound operation are shown in the image below.

Right rotation of child `Ch` about parent `Par`.

5.4.2 Global tree balancing (The DSW algorithm)

Example :



5.4.2 Global tree balancing (The DSW algorithm)

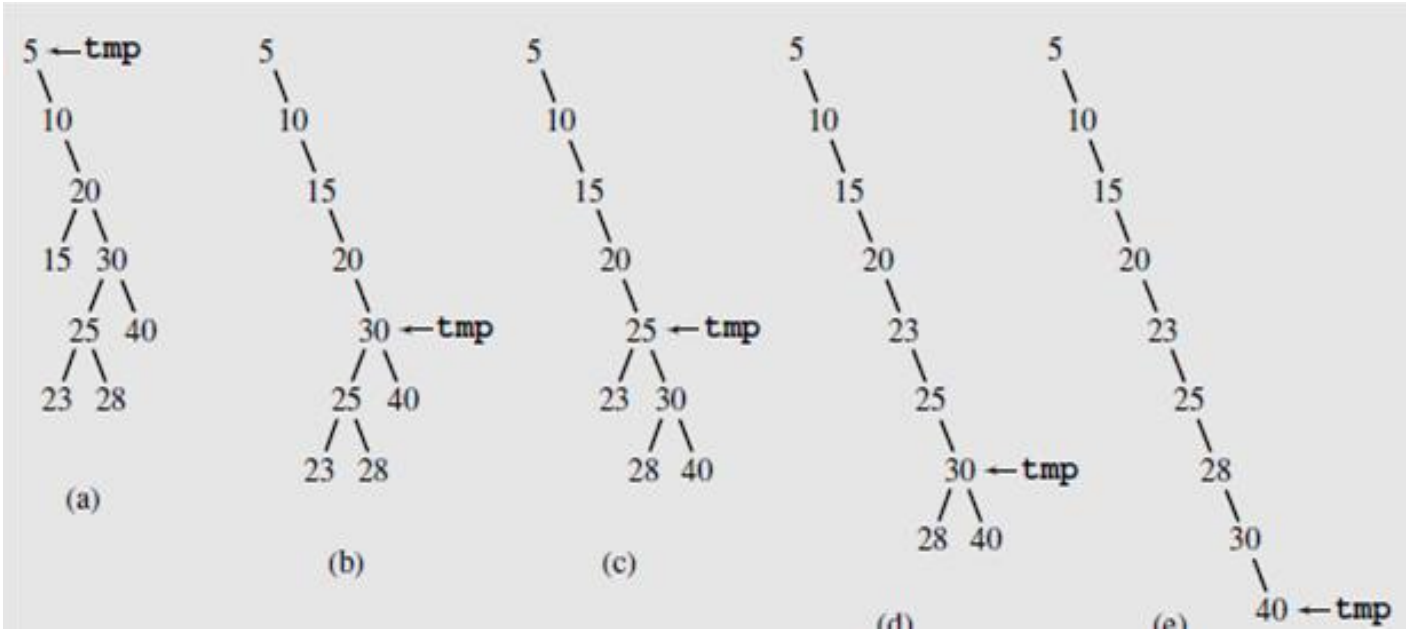
- So the idea is to take a tree and perform some rotations to it to make it balanced.
- First you create a backbone or a vine
- Then you transform the backbone into a nicely balanced tree

- createBackbone(root, n)
  - Tmp = root
  - While (Tmp != 0)
    - If Tmp has a left child
      - Rotate this child about Tmp
      - Set Tmp to the child which just became parent
    - Else set Tmp to its right child

- createPerfectTree(n)
  - $M = 2^{floor[lg(n+1)]}-1$;
  - Make n-M rotations starting from the top of the backbone;
  - While (M > 1)
    - M = M/2;
    - Make M rotations starting from the top of the backbone;

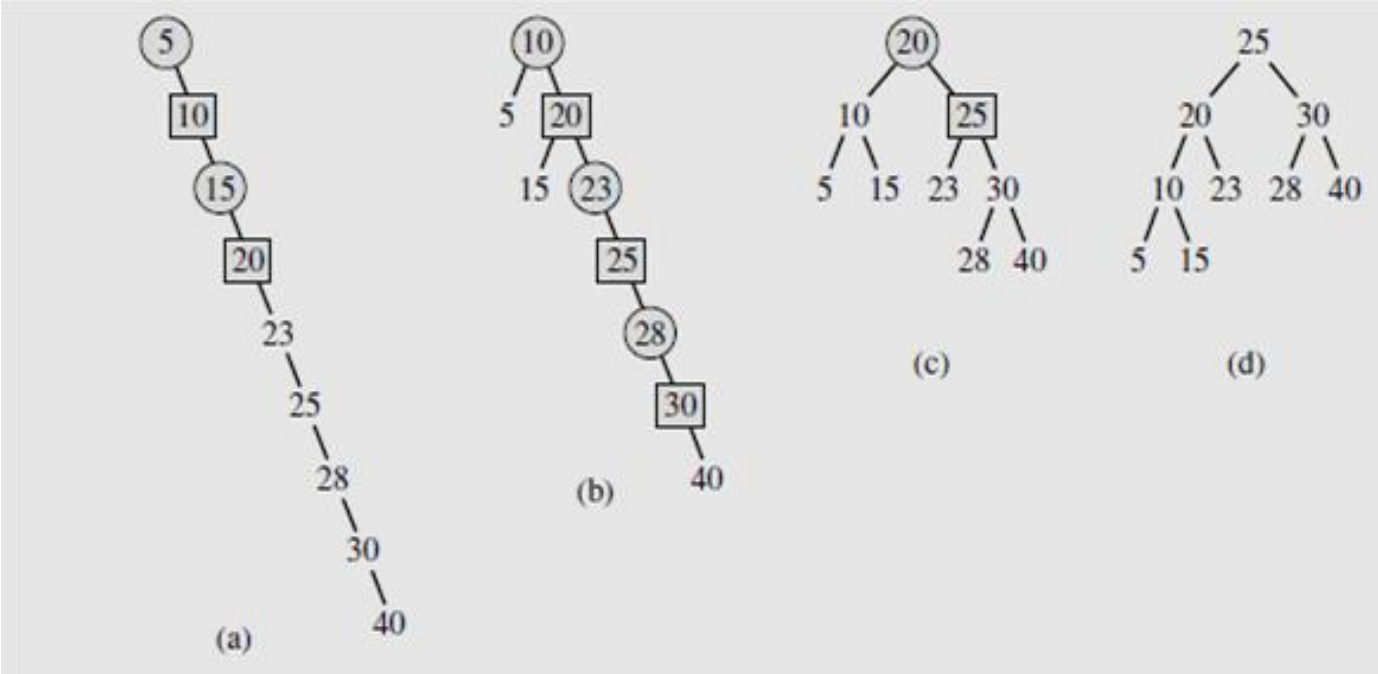
5.4.2 Global tree balancing (The DSW algorithm)

Transforming a binary search tree into a backbone.



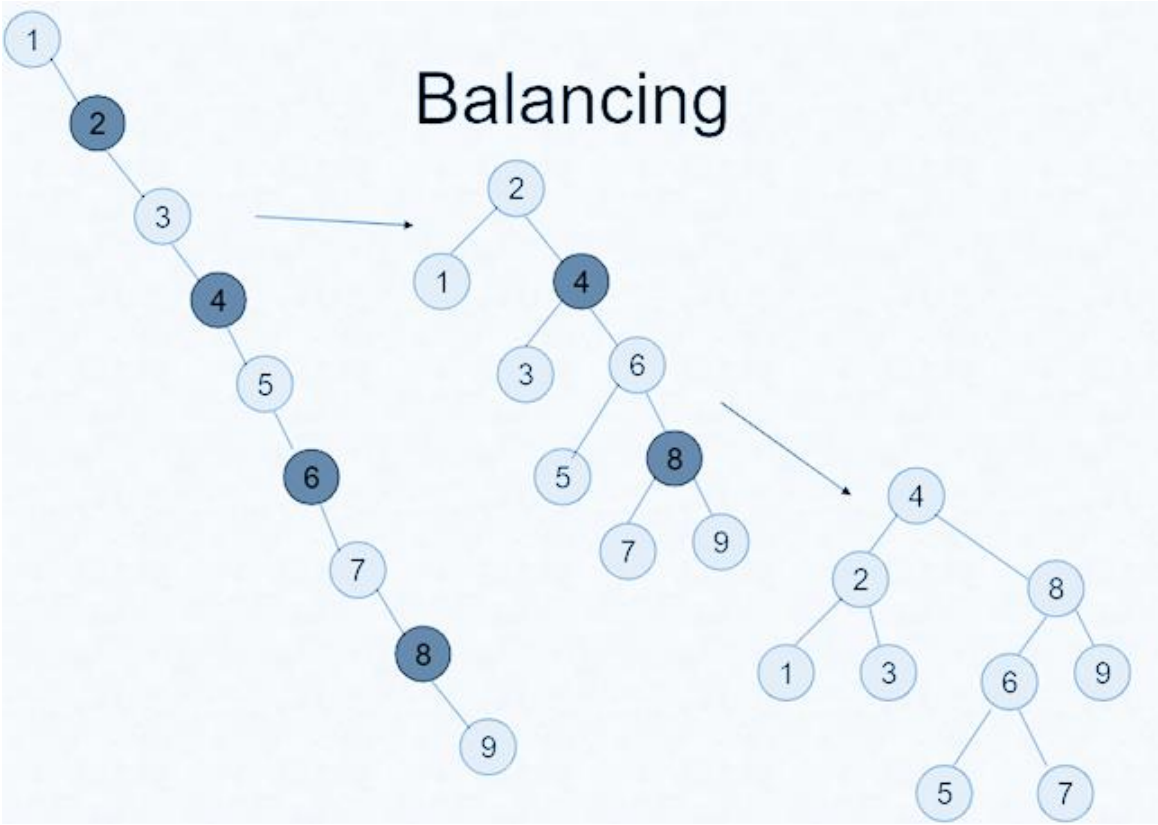
5.4.2 Global tree balancing (The DSW algorithm)

Transforming a backbone into a perfectly balanced tree.



5.4.2 Global tree balancing (The DSW algorithm)

Example :



5.4.2 Global tree balancing (The DSW algorithm)

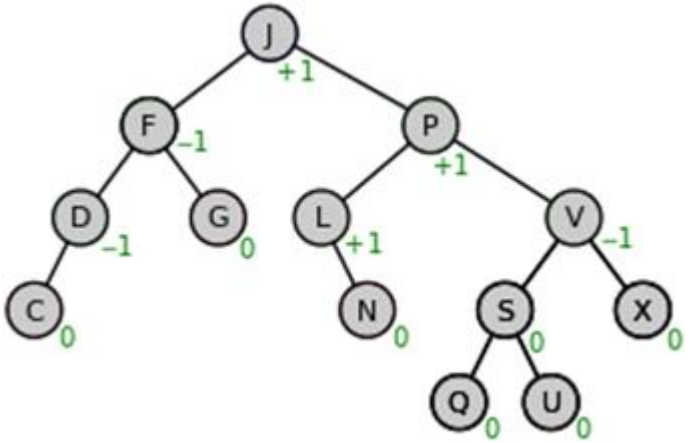
- The DSW algorithm is effective at balancing an entire tree. (Actually O(n))
- Sometimes trees need only be balanced periodically,in which case this cost can be amortised.
- Alternatively the tree may only become unstable after a series of insertions and deletions,in which case a DSW balancing may be appropriate.
- An alternative approach is to ensure the tree remains balanced by incorporating balancing into any insertion/deletion algorithms.
  - In an AVL insertion and deletion considers the structure of the tree
    - All nodes have a balance factor - i.e. a difference between the number of nodes in the right subtree and the left subtree.This difference must remain at +1,0 or -1
    - An AVL tree may not appear completely balanced as after the DSW algorithm

5.4.3 Local tree balancing (AVL Tree)

- Named after its inventors Adel'son-Vel'skii and Landis, hence AVL
- The heights of any subtree can only differ by at most one.
- Each nodes will indicate balance factors.
- Worst case for an AVL tree is 44% worst then a perfect tree.
- In practice, it is closer to a perfect tree.
- Each time the tree structure is changed, the balance factors are checked and if an imbalance is recognized, then the tree is restructured.
- For insertion there are four cases to be concerned with.
- Deletion is a little trickier.

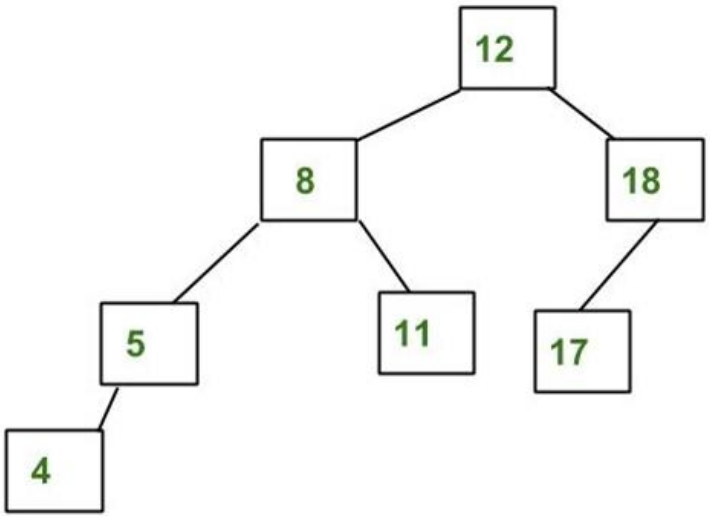
5.4.3 Local tree balancing (AVL Tree)

- Each time the tree structure is changed, the balance factors are checked and if an imbalance is recognized, then the tree is restructured.
- For insertion there are four cases to be concerned with.
- Deletion is a little trickier
- The balance factor of any node in the tree is -1 , o or +1. It implies that the tree is an AVL tree.
- Example :

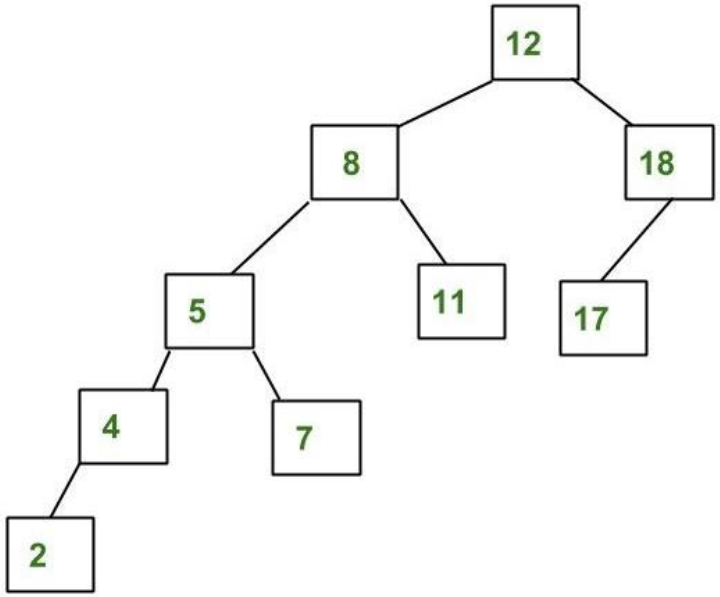


5.4.3 Local tree balancing (AVL Tree)

Examples for AVL and Non-AVL tree



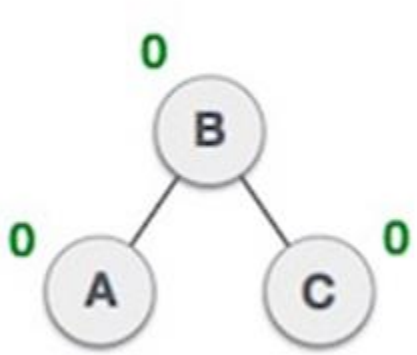
AVL Tree



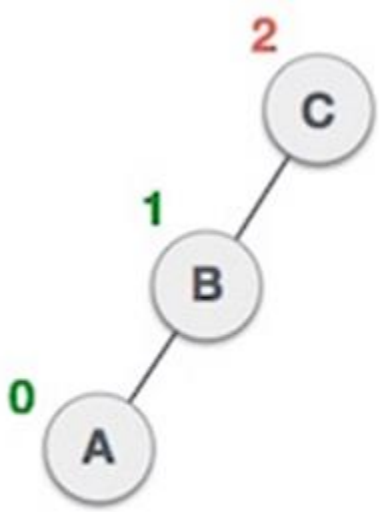
NOT an AVL Tree

5.4.3 Local tree balancing (AVL Tree)

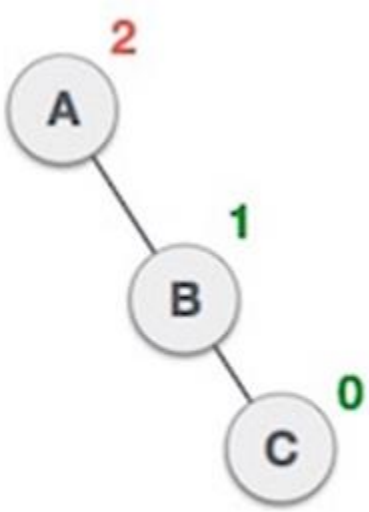
Examples for AVL and Non-AVL tree



Balanced



Not balanced



Not balanced

5.4.3 Local tree balancing (AVL Tree)

- **Algorithm for Inserting new nodes into an AVL trees**
  - Insert the node in the same way as in an ordinary binary search tree.
  - Beginning with the new node, trace a path back towards the root, checking the difference in height of the sub-trees at each node along the way.
  - If you find a node with an imbalance (**a height difference other than 0,+1,-1**), stop your trace at this point.
  - Consider the node with the imbalance and two nodes on the layers immediately below this point on the path back to the new node.
  - If these three nodes lie in a straight line, apply a single rotation to correct the imbalance.
  - If these three nodes lie in a dog-leg pattern, apply a double rotation to correct the imbalance.

5.4.3 Local tree balancing (AVL Tree)

- **Rotations**
  - When the tree structure changes (e.g., with insertion or deletion), we need to modify the tree to restore the AVL tree property.
  - This can be done using single rotations or double rotations. Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of a sub tree by 1.
  - So, if the AVL tree property is violated at a node X, it means that the heights of left(X) and right(X) differ by exactly 2. This is because, if we balance the AVL tree every time, then at any point, the difference in heights of left(X) and right(X) differ by exactly 2.
  - Rotations is the technique used for restoring the AVL tree property. This means, we need to apply the rotations for the node X.

5.4.3 Local tree balancing (AVL Tree)

- **Rotations**

The rules for deciding which type of rotations to be used.

1. When you have found the first node is out of balance, restrict your attention to that node and the two nodes in the two layers immediately below it.
2. If these three nodes lie in a straight line, single rotation is needed to restore the balance.
3. If these three nodes lie in a "**dog leg**" pattern you need double rotation to restore the balance.

5.4.3 Local tree balancing (AVL Tree)

- **Types of Violations**
  - Let us assume the node that must be rebalanced is X. Since any node has at most two children, and a height imbalance requires that X's two sub tree heights differ by two, we can observe that a violation might occur in four cases:
    1. An insertion into the left sub tree of the left child of X.
    2. An insertion into the right sub tree of the left child of X.
    3. An insertion into the left sub tree of the right child of X.
    4. An insertion into the right sub tree of the right child of X

Cases 1 and 4 are symmetric and easily solved with single rotations. Similarly, cases 2 and 3 are also symmetric and can be solved with double rotations

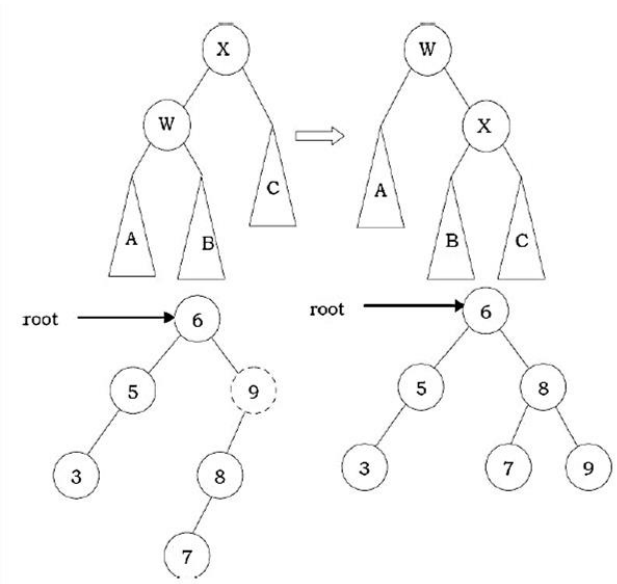
5.4.3 Local tree balancing (AVL Tree)

- **Single Rotations**

  - **Left Left Rotation (LL Rotation) [Case-1]:** In the case below, node X is not satisfying the AVL tree property. The rotation does not have to be done at the root of a tree.
  - In general, we start at the node inserted and travel up the tree, updating the balance information at every node on the path.

5.4.3 Local tree balancing (AVL Tree)

- **Single Rotations**
  - **Left Left Rotation (LL Rotation) [Case-1]:**

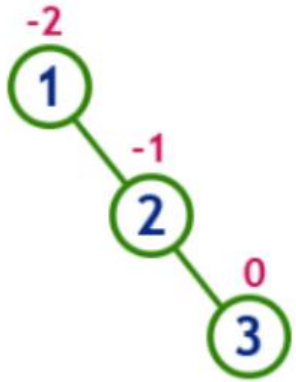


For example, in the figure above, after the insertion of 7 in the original AVL tree on the left, node 9 becomes unbalanced. So, we do a single left-left rotation at 9. As a result we get the tree on the right.

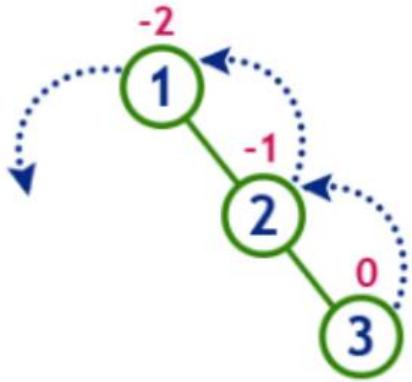
5.4.3 Local tree balancing (AVL Tree)

- **Single Rotations**
  - **Left Left Rotation (LL Rotation):**
    Example 2

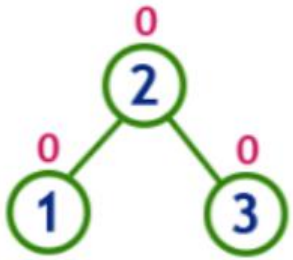
insert 1, 2 and 3



Tree is imbalanced



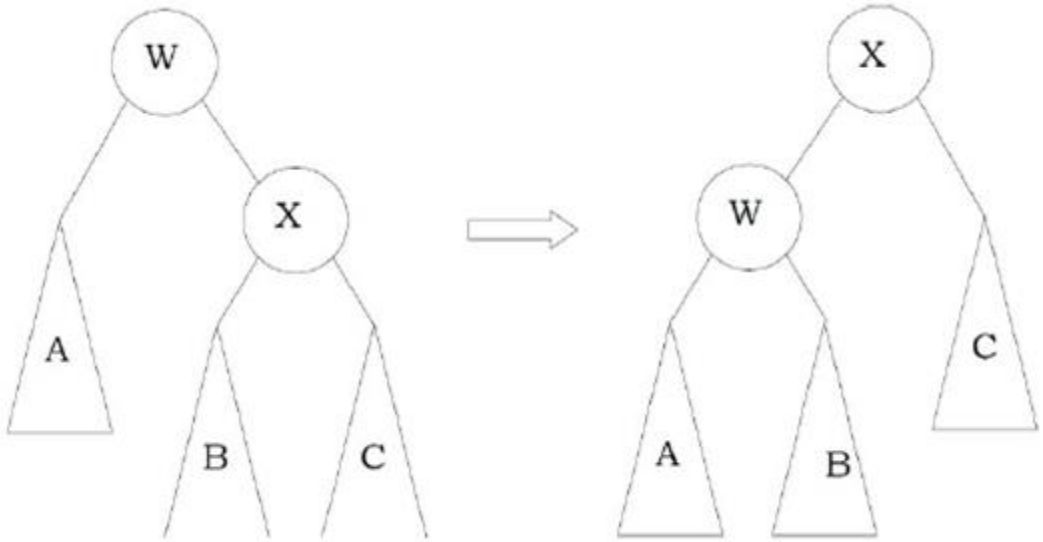
To make balanced we use LL Rotation which moves nodes one position to left



After LL Rotation Tree is Balanced

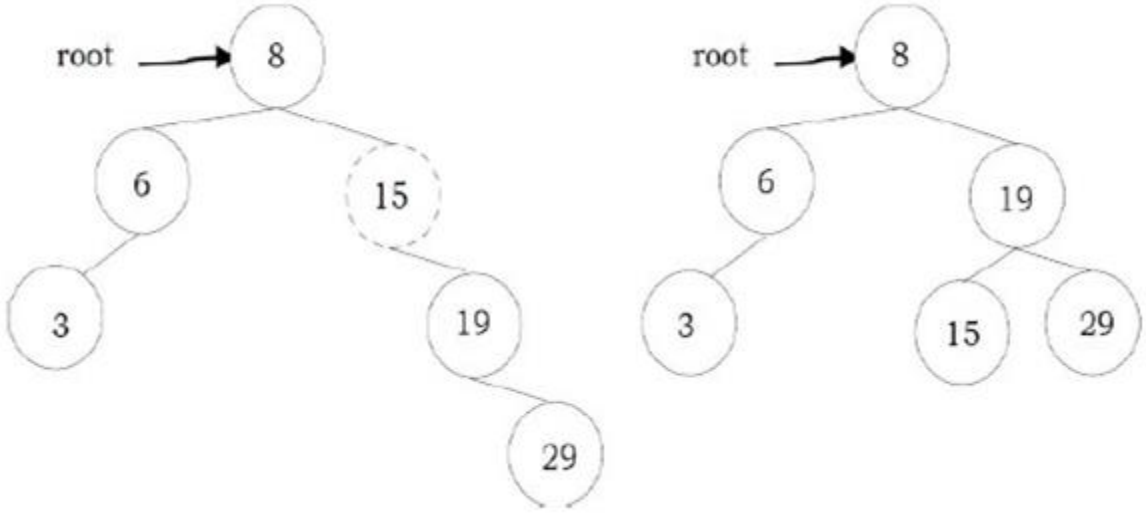
5.4.3 Local tree balancing (AVL Tree)

- **Single Rotations**
  - **Right Right Rotation (RR Rotation) [Case-4]:** In this case, node X is not satisfying the AVL tree property.



5.4.3 Local tree balancing (AVL Tree)

- **Single Rotations**
  - **Right Right Rotation (RR Rotation) [Case-4]:**

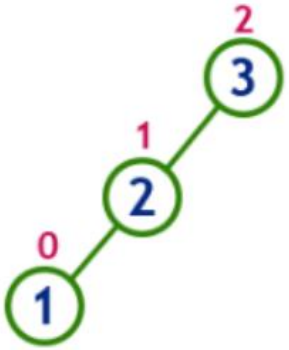


For example, in the figure, after the insertion of 29 in the original AVL tree on the left, node 15 becomes unbalanced. So, we do a single right-right rotation at 15. As a result we get the tree on the right.

5.4.3 Local tree balancing (AVL Tree)

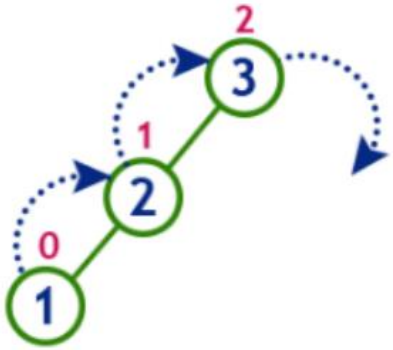
- **Single Rotations**
  - **Right Right Rotation (RR Rotation):**
    Example 2 :

insert 3, 2 and 1

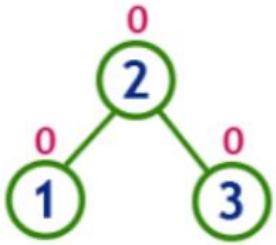


Tree is imbalanced

because node 3 has balance factor 2



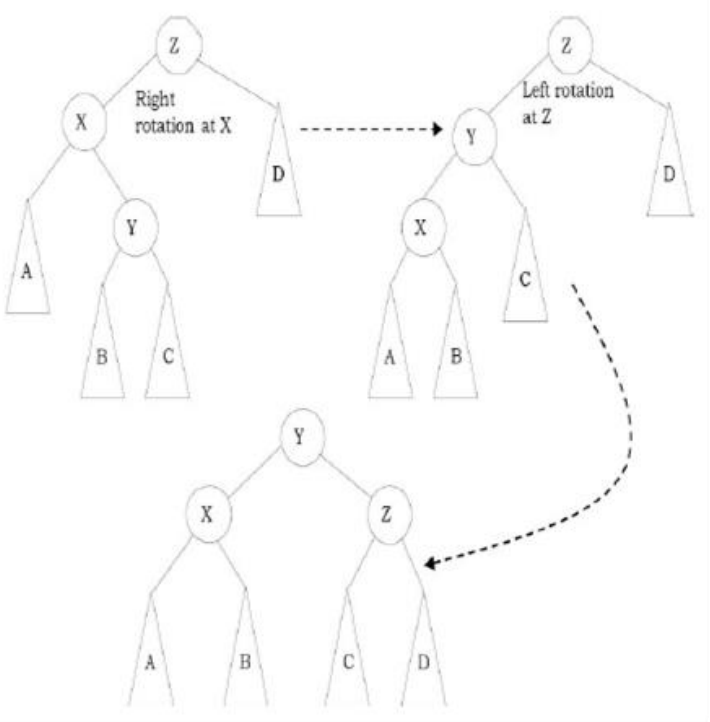
To make balanced we use RR Rotation which moves nodes one position to right



After RR Rotation Tree is Balanced

5.4.3 Local tree balancing (AVL Tree)

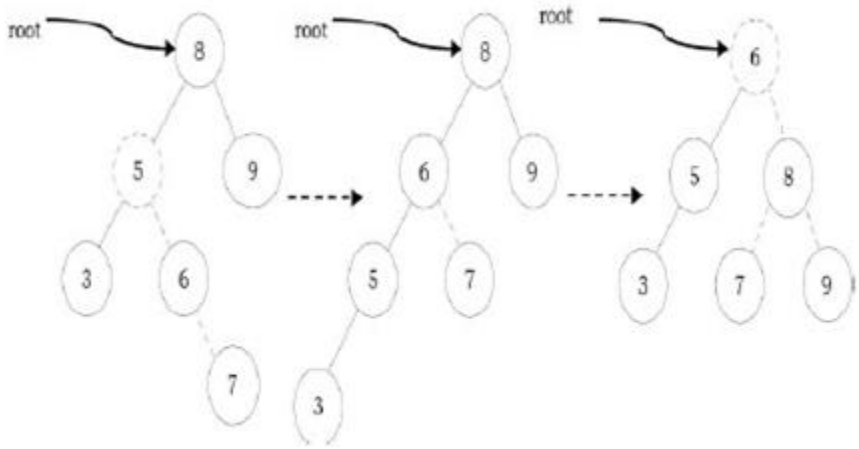
- **Double Rotations**
  - **Left Right Rotation (LR Rotation) [Case-2]:** For case-2 and case-3 single rotation does not fix the problem. We need to perform two rotations.



5.4.3 Local tree balancing (AVL Tree)

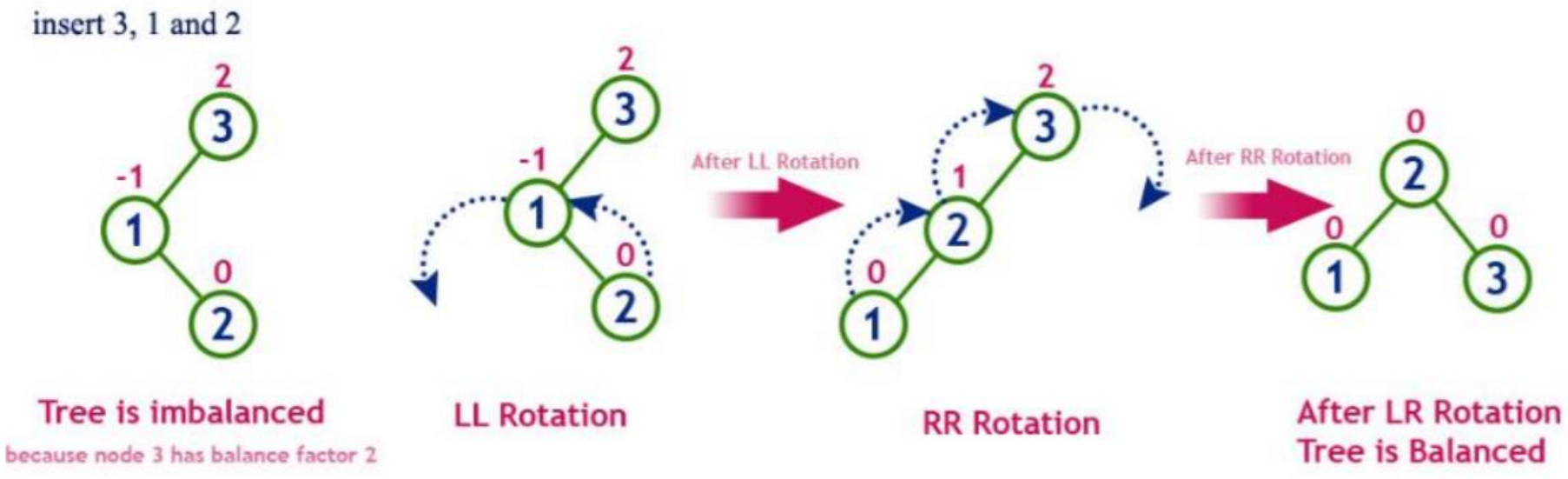
- **Double Rotations**
  - **Left Right Rotation (LR Rotation) [Case-2]:**

As an example, let us consider the following tree: The insertion of 7 is creating the case-2 scenario and the right side tree is the one after the double rotation.



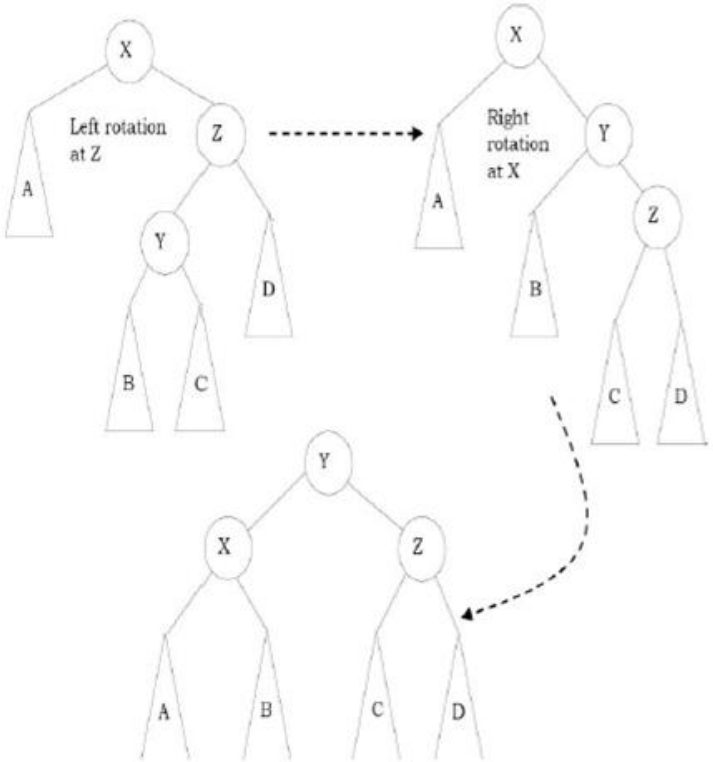
5.4.3 Local tree balancing (AVL Tree)

- **Double Rotations**
  - **Left Right Rotation (LR Rotation):**
    Example 2:



5.4.3 Local tree balancing (AVL Tree)

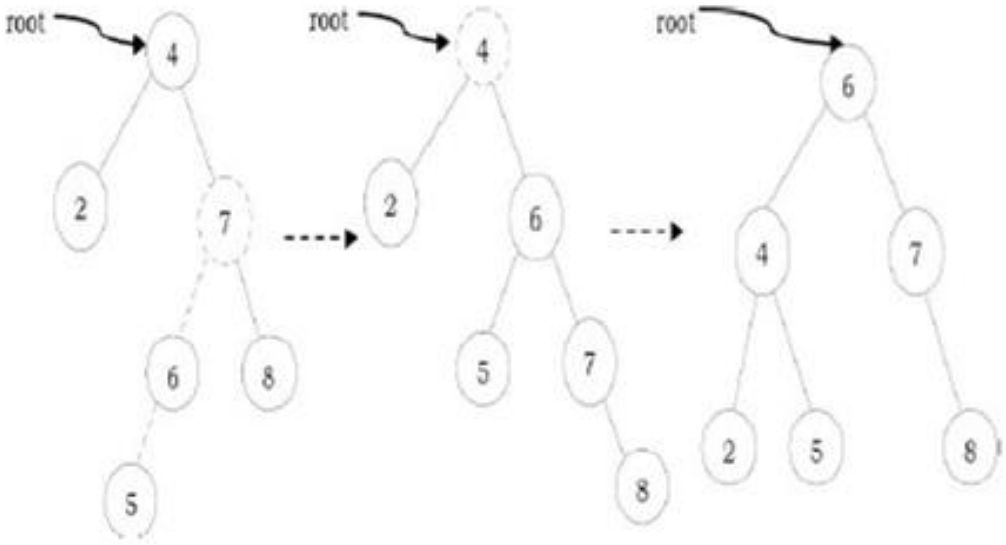
- **Double Rotations**
  - **Right Left Rotation (RL Rotation) [Case-3]:** Similar to case-2, we need to perform two rotations to fix this scenario.



5.4.3 Local tree balancing (AVL Tree)

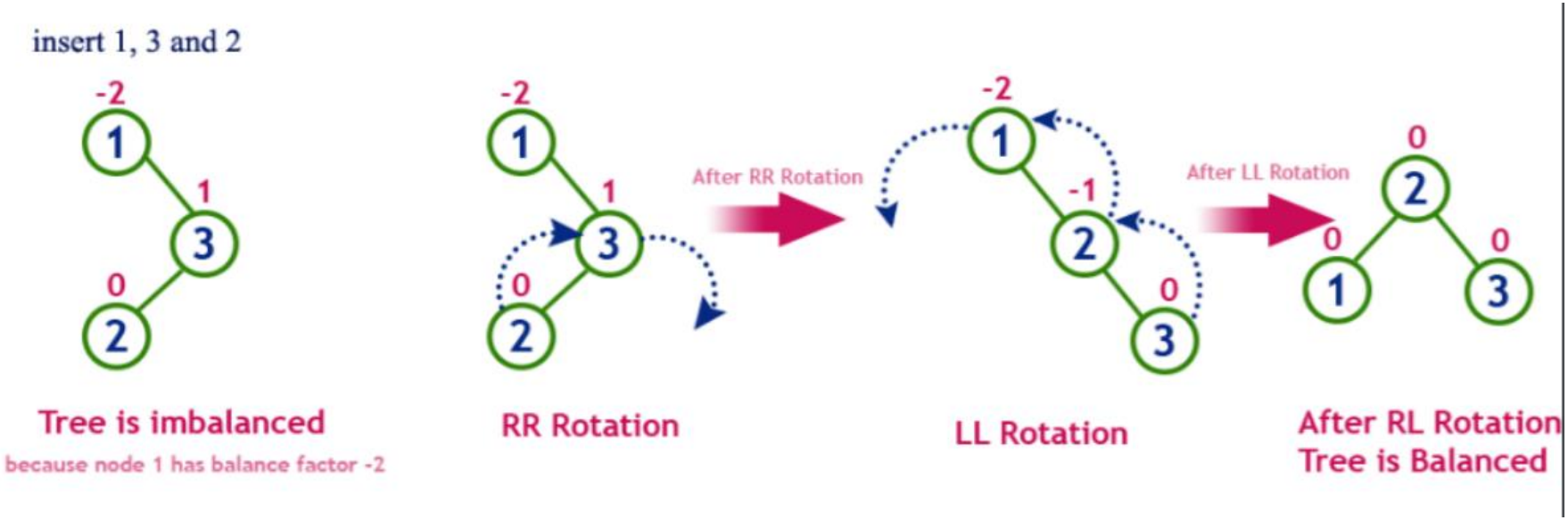
- **Double Rotations**
  - **Right Left Rotation (RL Rotation) [Case-3]:**

As an example, let us consider the following tree: The insertion of 6 is creating the case-3 scenario and the right side tree is the one after the double rotation.

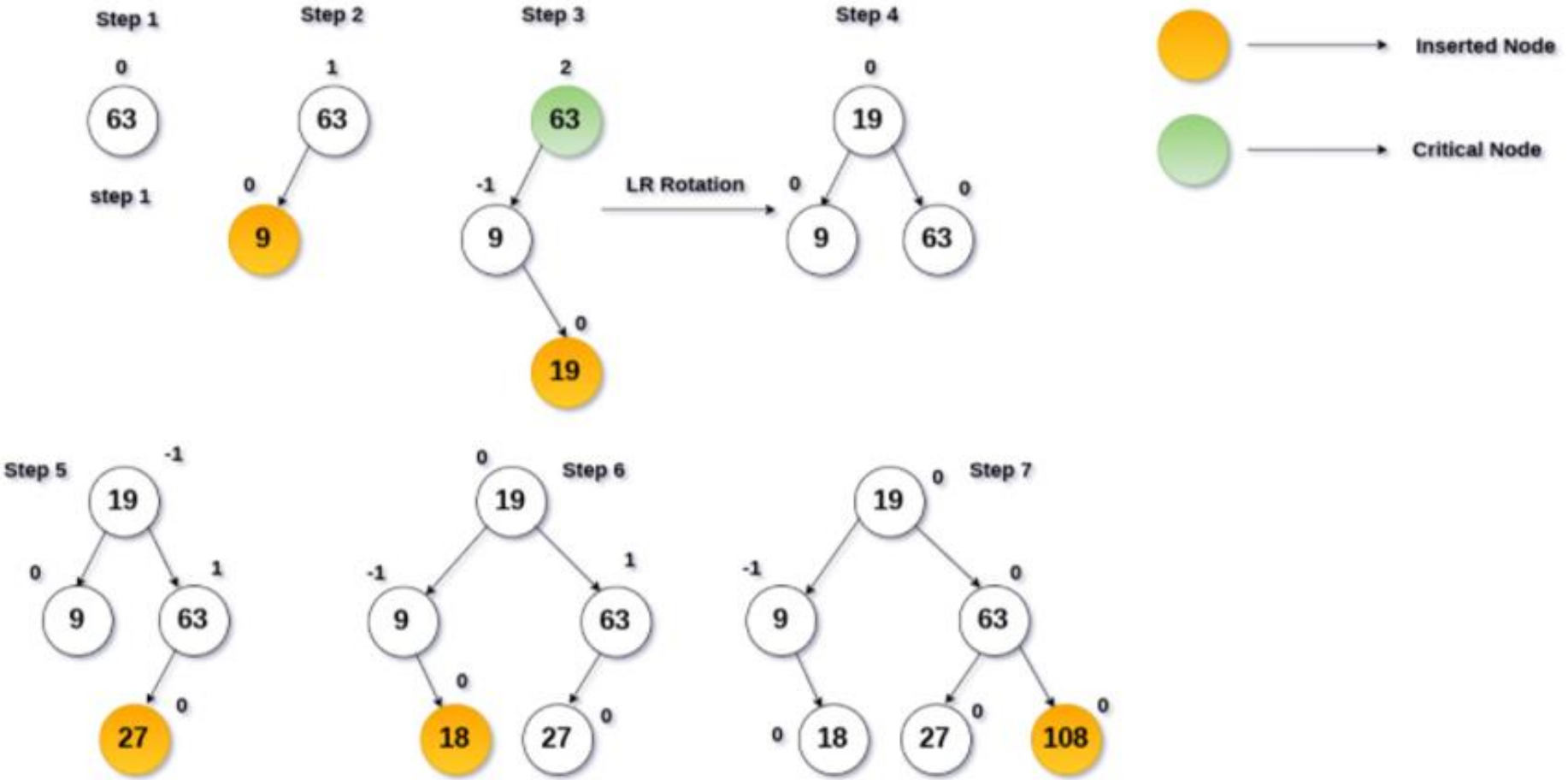


5.4.3 Local tree balancing (AVL Tree)

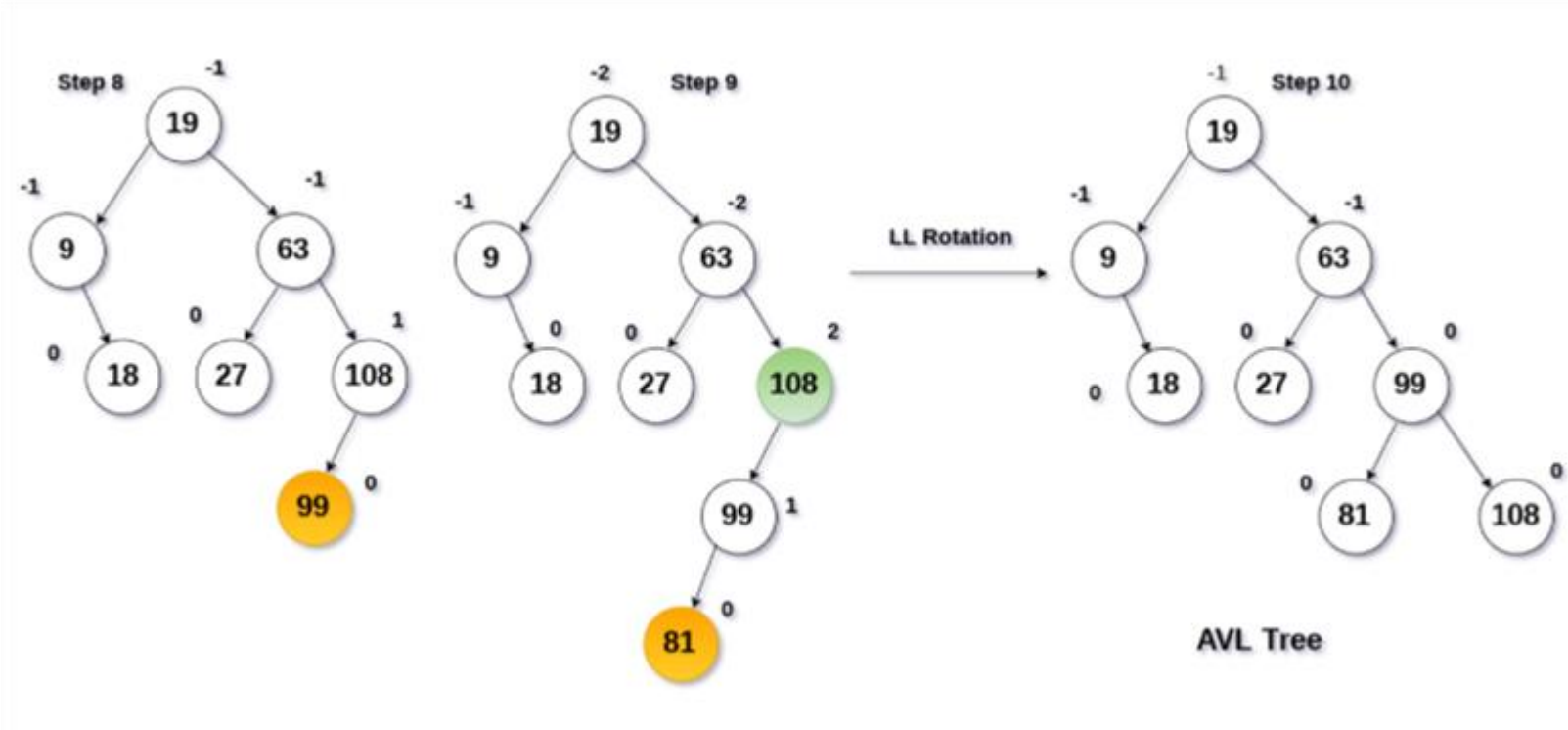
- **Double Rotations**
  - **Right Left Rotation (RL Rotation):**
    Example 2:



5.4.3 Local tree balancing (AVL Tree)

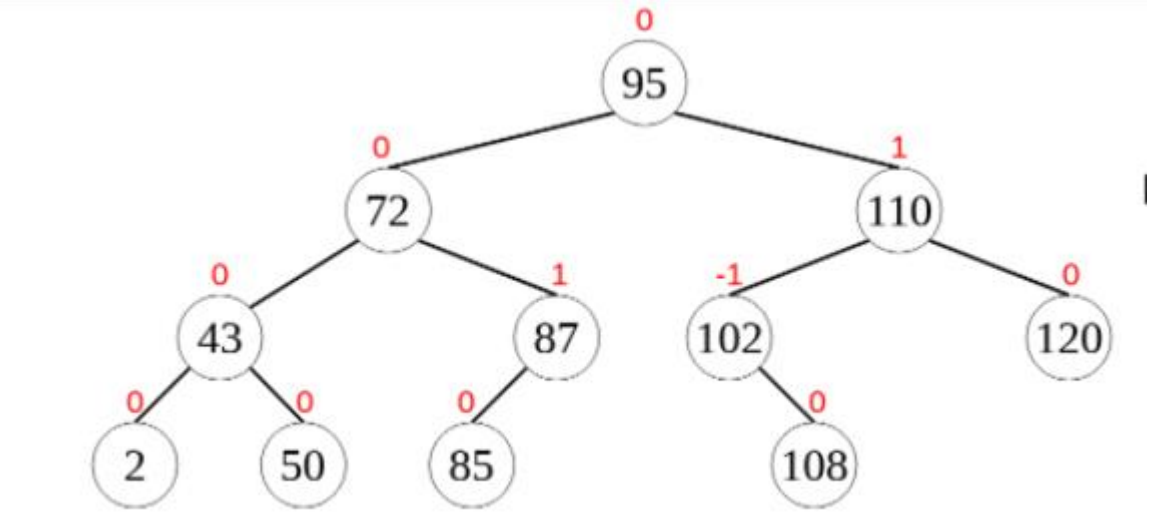


5.4.3 Local tree balancing (AVL Tree)



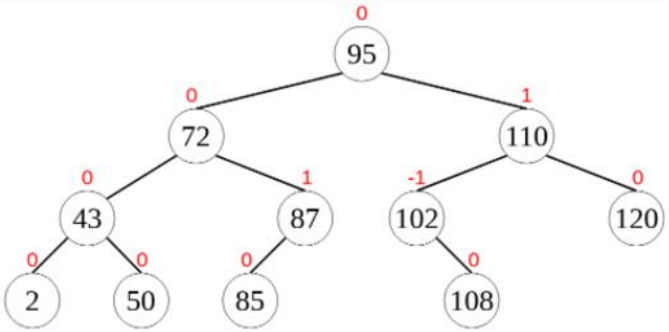
5.4.3 Local tree balancing (AVL Tree)

Example : Add a new node 105 to the below tree



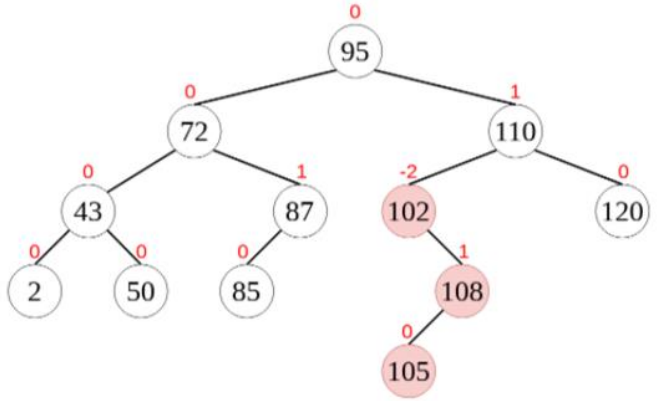
5.4.3 Local tree balancing (AVL Tree)

Example : Add a new node 105 to the below tree



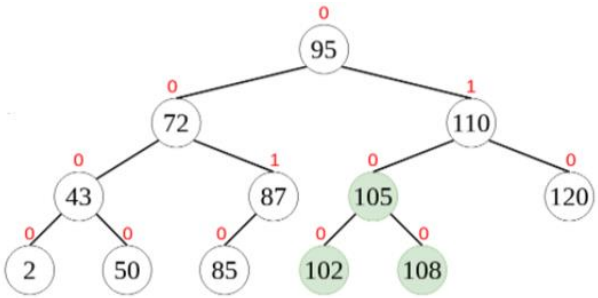
The tree is balanced

Insert 105



The tree is not balanced

Do RL rotation



The tree is balanced

5.4.3 Local tree balancing (AVL Tree)

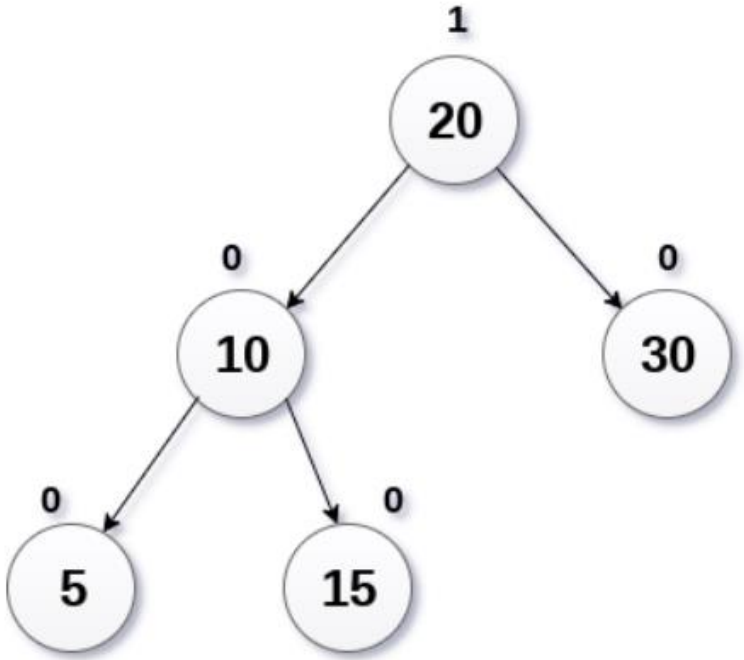
- Deletion is a bit trickier.
- With insertion after the rotation we were done.
- Not so with deletion.
- We need to continue checking balance factors as we travel up the tree
- Go ahead and delete the node just like in a BST.
- There are 4 cases after the deletion:

5.4.3 Local tree balancing (AVL Tree)

- **Case 1**: Deletion from a left subtree from a tree with a right high root and a right high right subtree.
  - Requires one left rotation about the root
- **Case 2**: Deletion from a left subtree from a tree with a right high root and a balanced right subtree.
  - Requires one left rotation about the root
- **Case 3**: Deletion from a left subtree from a tree with a right high root and a left high right subtree with a left high left subtree.
  - Requires a right rotation around the right subtree root and then a left rotation about the root
- **Case 4**: Deletion from a left subtree from a tree with a right high root and a left high right subtree with a right high left subtree
  - Requires a right rotation around the right subtree root and then a left rotation about the root

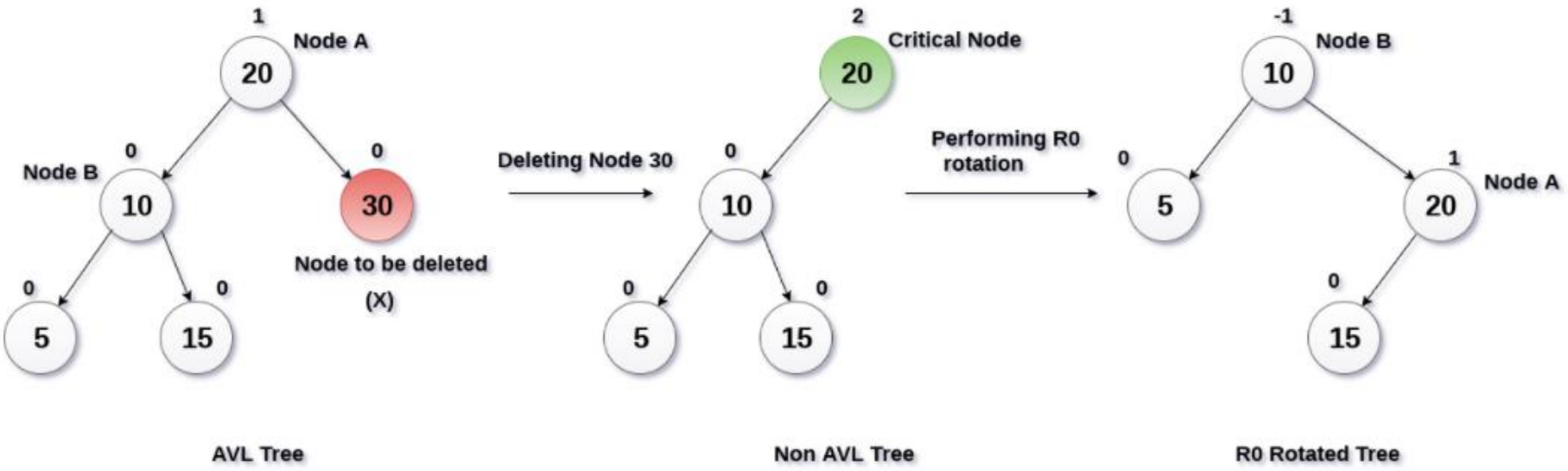
5.4.3 Local tree balancing (AVL Tree)

Example 1 : Delete node 30



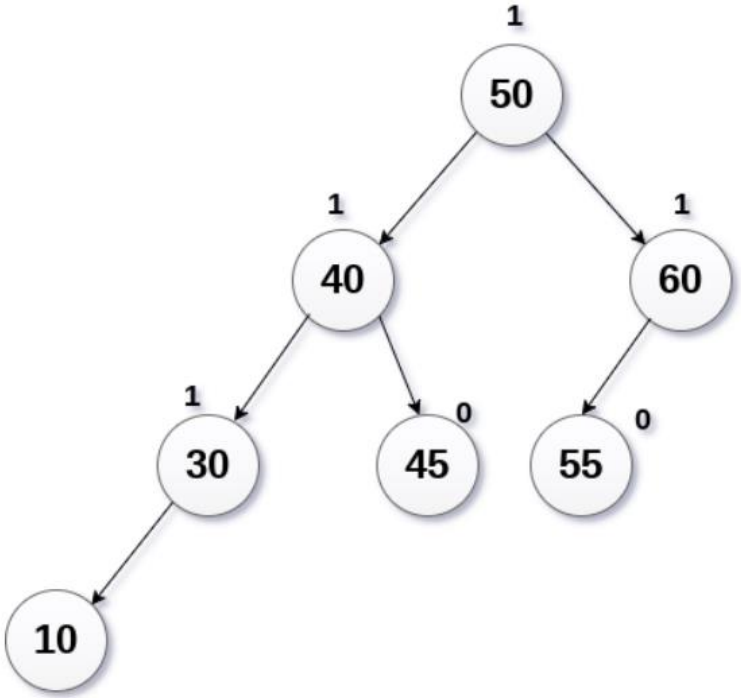
5.4.3 Local tree balancing (AVL Tree)

Example 1 : Delete node 30



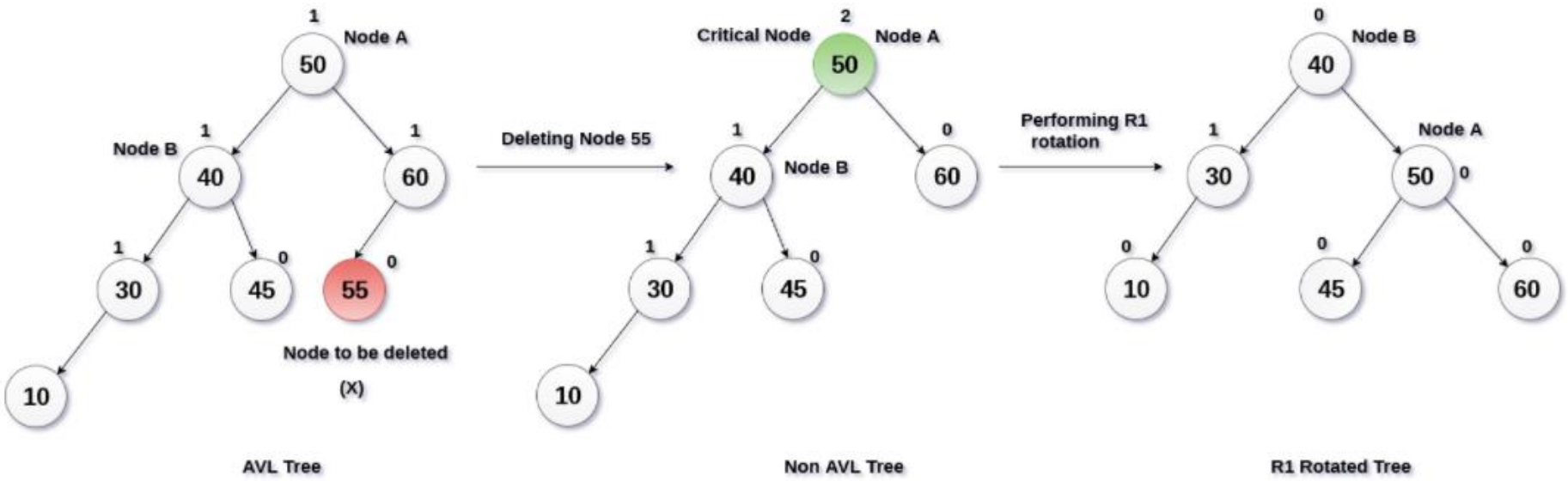
5.4.3 Local tree balancing (AVL Tree)

Example 2 : Delete node 55



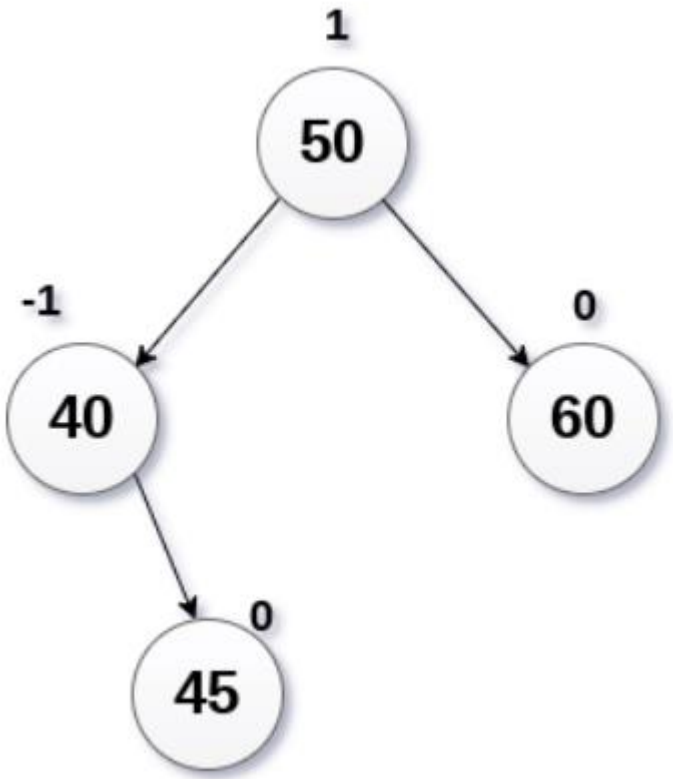
5.4.3 Local tree balancing (AVL Tree)

Example 2 : Delete node 55



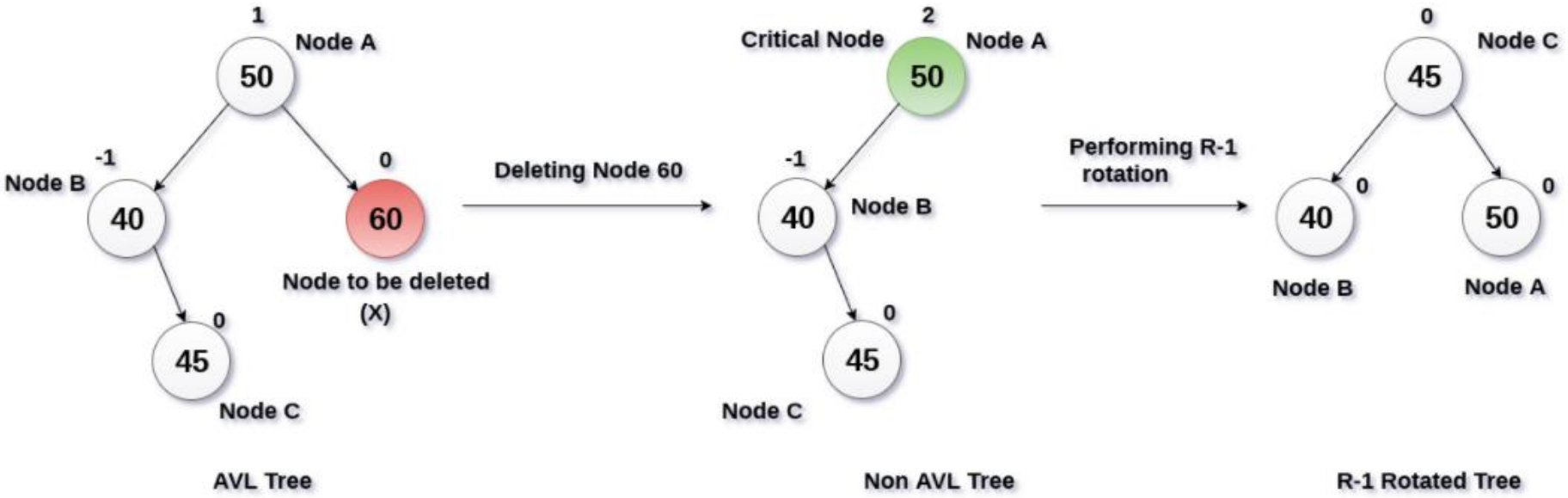
5.4.3 Local tree balancing (AVL Tree)

Example 3 : Delete node 60



5.4.3 Local tree balancing (AVL Tree)

Example 3 : Delete node 60



5.4.3 Local tree balancing (AVL Tree)

Code Segment :

```
Node rotateRight(Node y) {
    Node x = y.left;
    Node z = x.right;
    x.right = y;
    y.left = z;
    updateHeight(y);
    updateHeight(x);
    return x;
}
```

5.4.3 Local tree balancing (AVL Tree)

```
Node rotateLeft(Node y) {
    Node x = y.right;
    Node z = x.left;
    x.left = y;
    y.right = z;
    updateHeight(y);
    updateHeight(x);
    return x;
}
```

5.4.3 Local tree balancing (AVL Tree)

```
Node rebalance(Node z) {
    updateHeight(z);
    int balance = getBalance(z);
    if (balance > 1) {
        if (height(z.right.right) >
  height(z.right.left)) {
            z = rotateLeft(z);
        } else {
            z.right = rotateRight(z.right);
            z = rotateLeft(z);
        }
```

5.4.3 Local tree balancing (AVL Tree)

```
    } else if (balance < -1) {
        if (height(z.left.left) >
  height(z.left.right))
            z = rotateRight(z);
        else {
            z.left = rotateLeft(z.left);
            z = rotateRight(z);
        }
    }
    return z;
}
```

5.4.3 Local tree balancing (AVL Tree)

```
Node insert(Node node, int key) {
    if (node == null) {
        return new Node(key);
    } else if (node.key > key) {
        node.left = insert(node.left, key);
    } else if (node.key < key) {
        node.right = insert(node.right, key);
    } else {
        throw new RuntimeException("duplicate Key!");
    }
    return rebalance(node);
}
```

5.4.3 Local tree balancing (AVL Tree)

```
Node delete(Node node, int key) {
    if (node == null) {
        return node;
    } else if (node.key > key) {
        node.left = delete(node.left, key);
    } else if (node.key < key) {
        node.right = delete(node.right, key);
    } else {
        if (node.left == null || node.right == null) {
            node = (node.left == null) ? node.right :
  node.left;
        }
```

5.4.3 Local tree balancing (AVL Tree)

```
else {
            Node mostLeftChild =
  mostLeftChild(node.right);
            node.key = mostLeftChild.key;
            node.right = delete(node.right, node.key);
        }
    }
    if (node != null) {
        node = rebalance(node);
    }
    return node;
}
```

5.5 Heaps

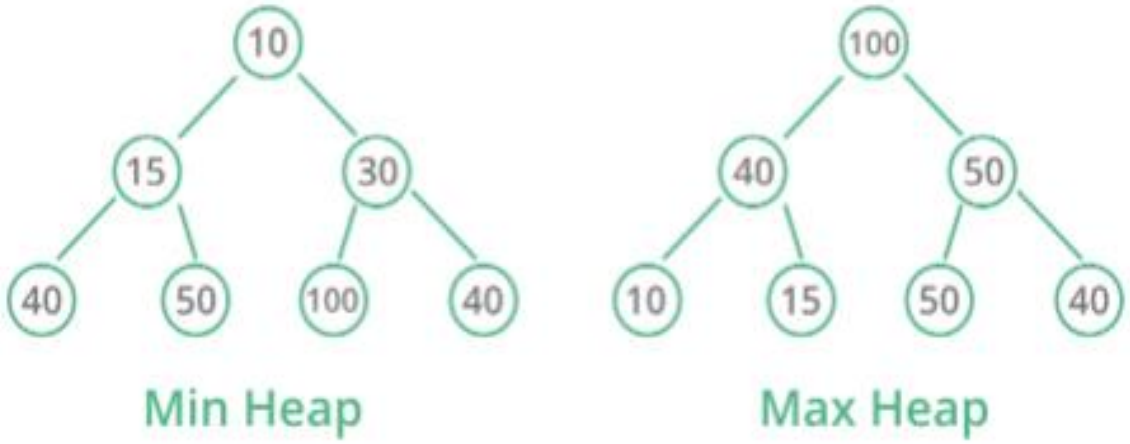
- A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

    - Max-Heap: In a Max-Heap the key present at the root node must be greatest among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree.

    - Min-Heap: In a Min-Heap the key present at the root node must be minimum among the keys present at all of it's children. The same property must be recursively true for all sub-trees in that Binary Tree

5.5 Heaps

- Example :



5.5 Heaps

Binary Heap

- A Binary Heap is a Binary Tree with following properties.

    - It's a complete tree (All levels are completely filled except possibly the last level and the last level has all keys as left as possible). This property of Binary Heap makes them suitable to be stored in an array.

    - A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

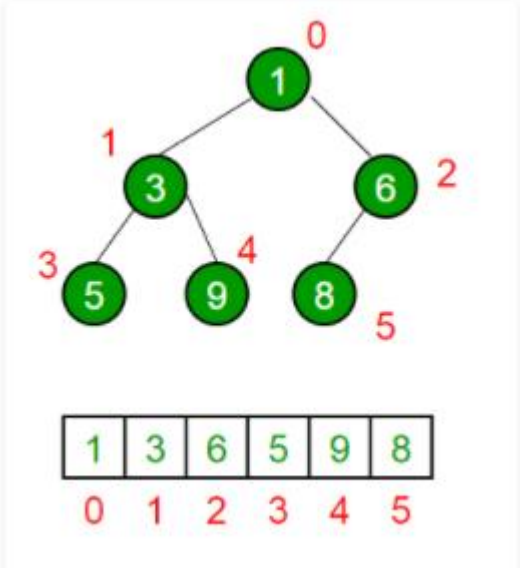
5.5 Heaps

- **Examples of Min Heap:**

5.5 Heaps

- **How is Binary Heap represented?**

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as an array.



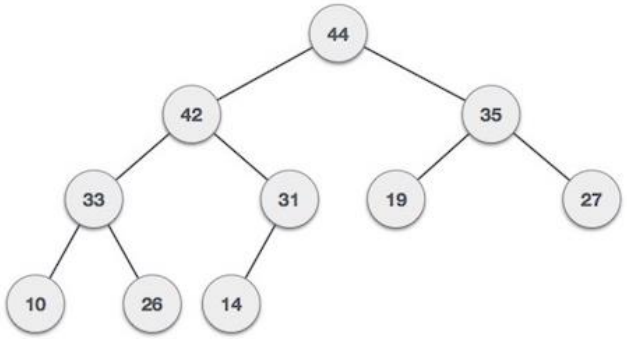
5.5 Heaps

- **Applications of Heaps**
  - Heap Data Structure is generally taught with Heapsort. Heapsort algorithm has limited uses because Quicksort is better in practice. Nevertheless, the Heap data structure itself is enormously used. Following are some uses other than Heapsort.
  - Priority Queues: Priority queues can be efficiently implemented using Binary Heap because it supports insert(), delete() and extractmax(), decreaseKey() operations in O(logn) time. Binomoial Heap and Fibonacci Heap are variations of Binary Heap.

5.5 Heaps

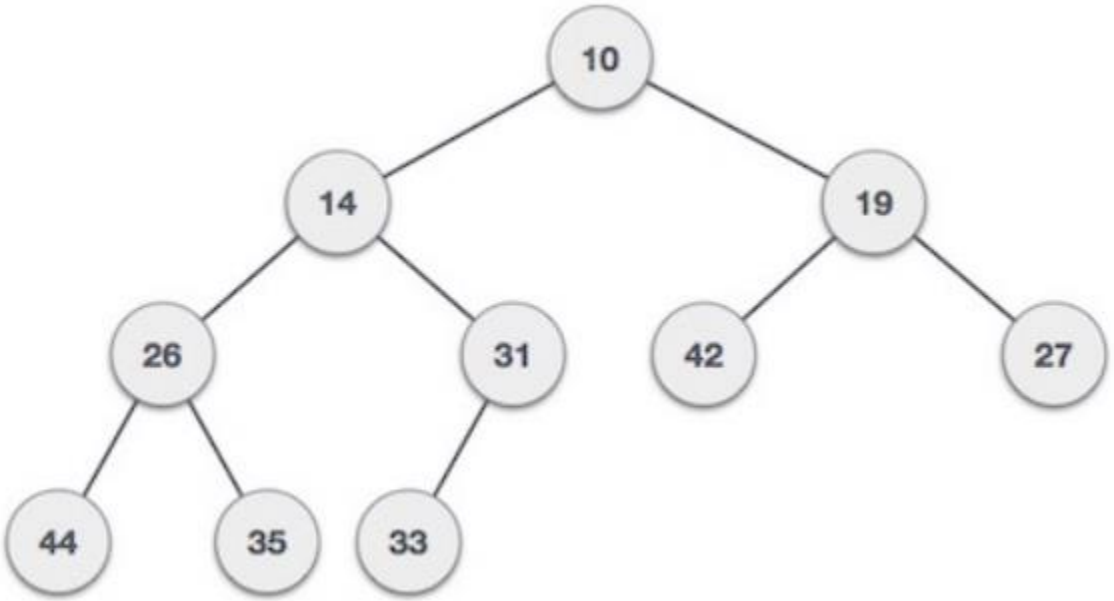
- Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If α has child node β then −

  key(α) ≥ key(β)
- As the value of parent is greater than that of child, this property generates Max Heap.
- Based on this criteria, a heap can be of two types
- **Max-Heap** − Where the value of the root node is greater than or equal to either of its children



5.5 Heaps

- **Min-Heap** – Where the value of the root node is less than or equal to either of its children



5.5 Heaps

- **Max Heap Construction Algorithm**

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

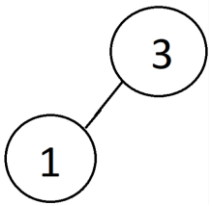
Step 5 − Repeat step 3 & 4 until Heap property holds.

5.5 Heaps

- Example : 3,1,6,5,2,4

  First Insert 3 in root of the empty heap:

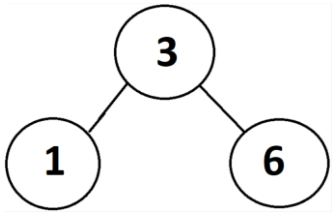
Next Insert 1 at the bottom of the heap. No need to swap the child node (1) with the parent node (3) , because 3 is greater than 1.



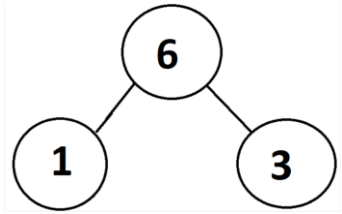
5.5 Heaps

- Example : 3,1,6,5,2,4

  Next insert 6 to the bottom of the heap, and since 6 is greater than 3, swapping is needed



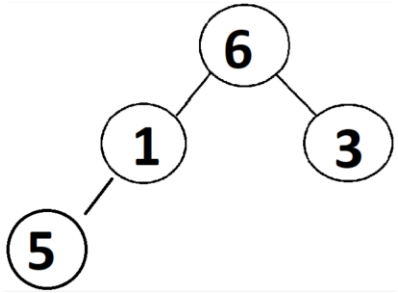
  Swap the 6 and the 3



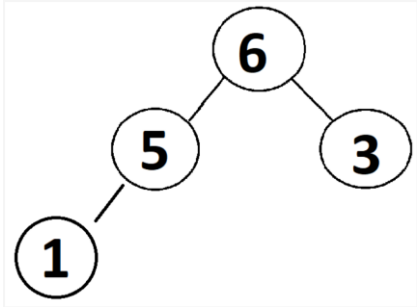
5.5 Heaps

- Example : 3,1,6,5,2,4

  Next insert 5 to the bottom of the heap, and since 5 is greater than 1, swapping is needed



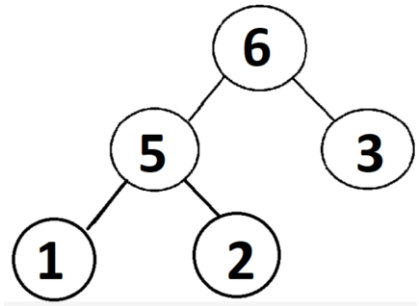
  Swap the 5 and the 1. Since 5 is less than 6, so no further swapping needed.



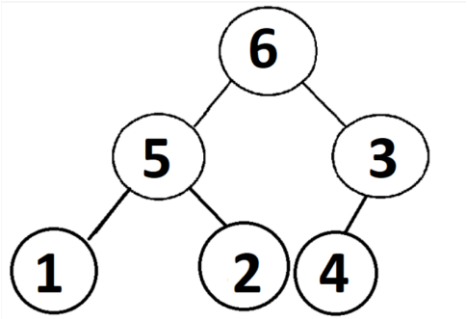
5.5 Heaps

- Example : 3,1,6,5,2,4

  Next insert 2 to the bottom of the heap, and since 2 is less than 5, no need to swap the 5 with the 2.



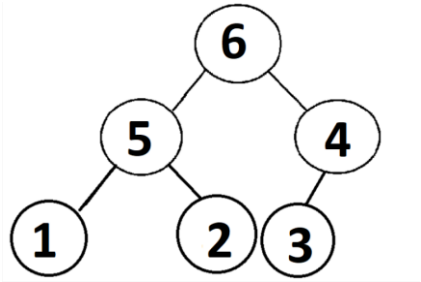
  Next insert 4 to the bottom of the heap, and since 4 is greater than 3, we need to swap the 4 and the 3.



5.5 Heaps

- Example : 3,1,6,5,2,4

  Swap the 4 and the 3. Since 4 is less than 6 no further swapping needed, and we are done.



The array will now look like the below:

6	5	4	1	2	3

5.5 Heaps

- **Max Heap Deletion Algorithm**

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

Step 1 – Remove root node.

Step 2 – Move the last element of last level to root.

Step 3 – Compare the value of this child node with its parent.

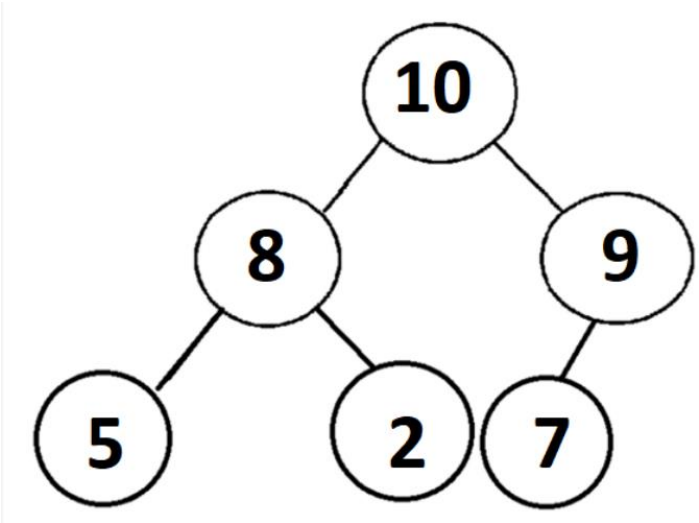
Step 4 – If value of parent is less than child, then swap them.

Step 5 – Repeat step 3 & 4 until Heap property holds.

5.5 Heaps

- Example :

10	8	9	5	2	7

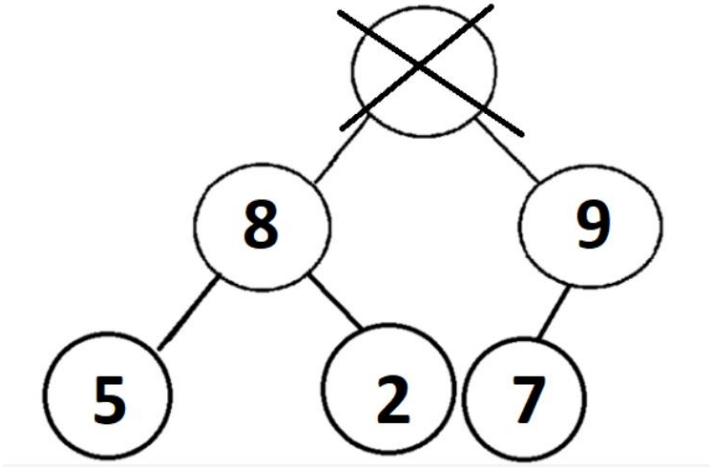


5.5 Heaps

- Example :

  Delete node 10

10	8	9	5	2	7

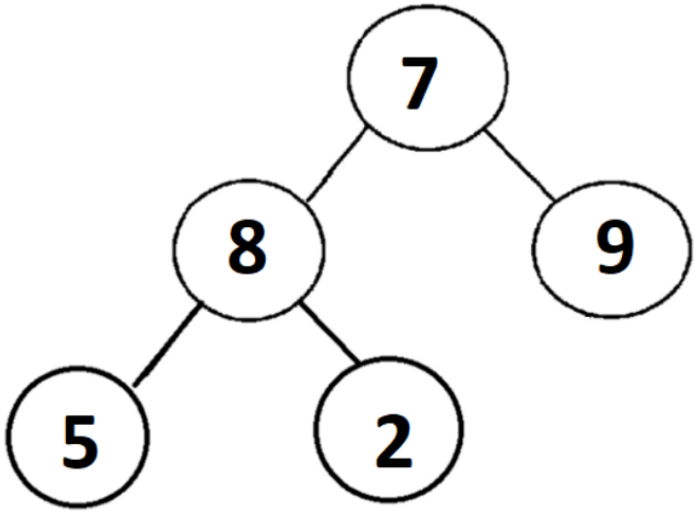


5.5 Heaps

- Example :

10	8	9	5	2	7

Replace the deleted node with the farthest right node.

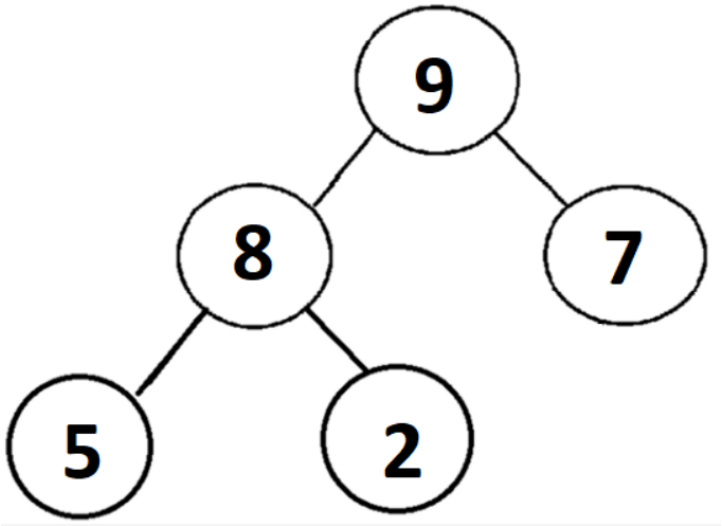


5.5 Heaps

- Example :

10	8	9	5	2	7

Heapify (Fix the heap):



Now the heap property holds true and we are done !

5.5 Heaps

Code Segment :

```
public class deletionHeap {
     static void heapify(int arr[], int n, int i)
    {
        int largest = i; // Initialize largest as root
        int l = 2 * i + 1; // left = 2*i + 1
        int r = 2 * i + 2; // right = 2*i + 2
        if (l < n && arr[l] > arr[largest])
            largest = l;
        if (r < n && arr[r] > arr[largest])
            largest = r;
        if (largest != i) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;
             heapify(arr, n, largest);
        }
    }
```

5.5 Heaps

Code Segment :

```
static int deleteRoot(int arr[], int n)
    {
        int lastElement = arr[n - 1];
       arr[0] = lastElement;
        n = n - 1;
        heapify(arr, n, 0);
        return n;
    }
    static void printArray(int arr[], int n)
        {
            for (int i = 0; i < n; ++i)
                System.out.print(arr[i] + " ");
                    System.out.println();
        }
```

5.5 Heaps

Code Segment :

```
    public static void main(String args[])
        {

            int arr[] = { 10, 5, 3, 2, 4 };

            int n = arr.length;

            n = deleteRoot(arr, n);

            printArray(arr, n);
        }
    }
```

Summary

Tree	A tree can be defined non recursively as a set of nodes and a set of directed edges that connect them
Binary Tree	An important use of binary trees is in other data structures, notably the binary search tree and the priority queue.
AVL Tree	The AVL tree was the first balanced binary search tree. It has historical significance and also illustrates most of the ideas that are used in other schemes