CHAPTER 6

DATA STRUCTURES AND ALGORITHMS

Tree
Data Structure

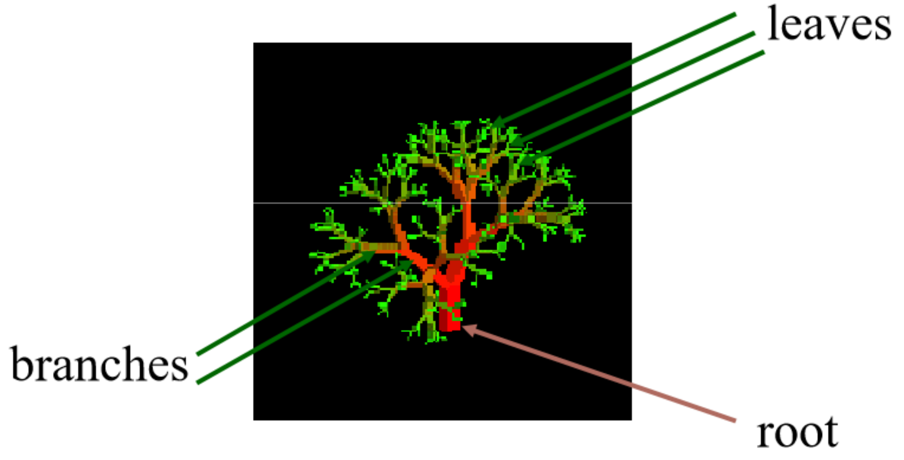
CONTENTS

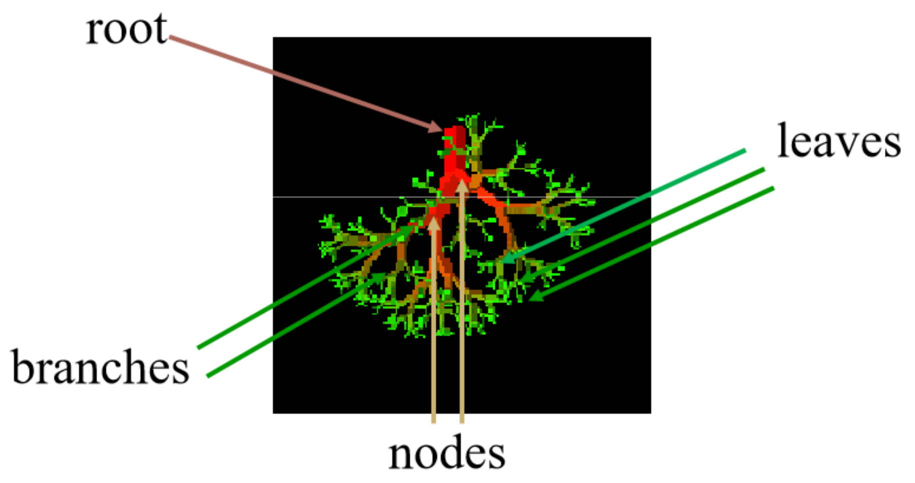
- Tree Data Structure:
  - Definition of tree,
  - Basic terminologies,
- Types of trees:-
  - n-ary tree, Binary tree, BST, AVL tree, full BT, complete BT ,Balanced BT .
- Basic operations on tree,
- Tree traversal methods:
  - in-order, pre-order, post-order
- **Heap data Structure:-** definition, creation, insertion, update, deletion, print etc. Examples of Expression trees

6.1. TREE DATA STRUCTURE

- So far we have seen
  - arrays, linked list, stacks, queues
- Disadvantages ?
  - They are linear structures (Time is Money)
- Tree
  - Non-linear structures (every data element may have more than one predecessor as well as successor)
  - Probably the most fundamental structure in computing
  - Hierarchical structure
  - Tree consist of nodes and edges (arcs)
- Applications:
  - Organization charts
  - File systems
  - Programming environments

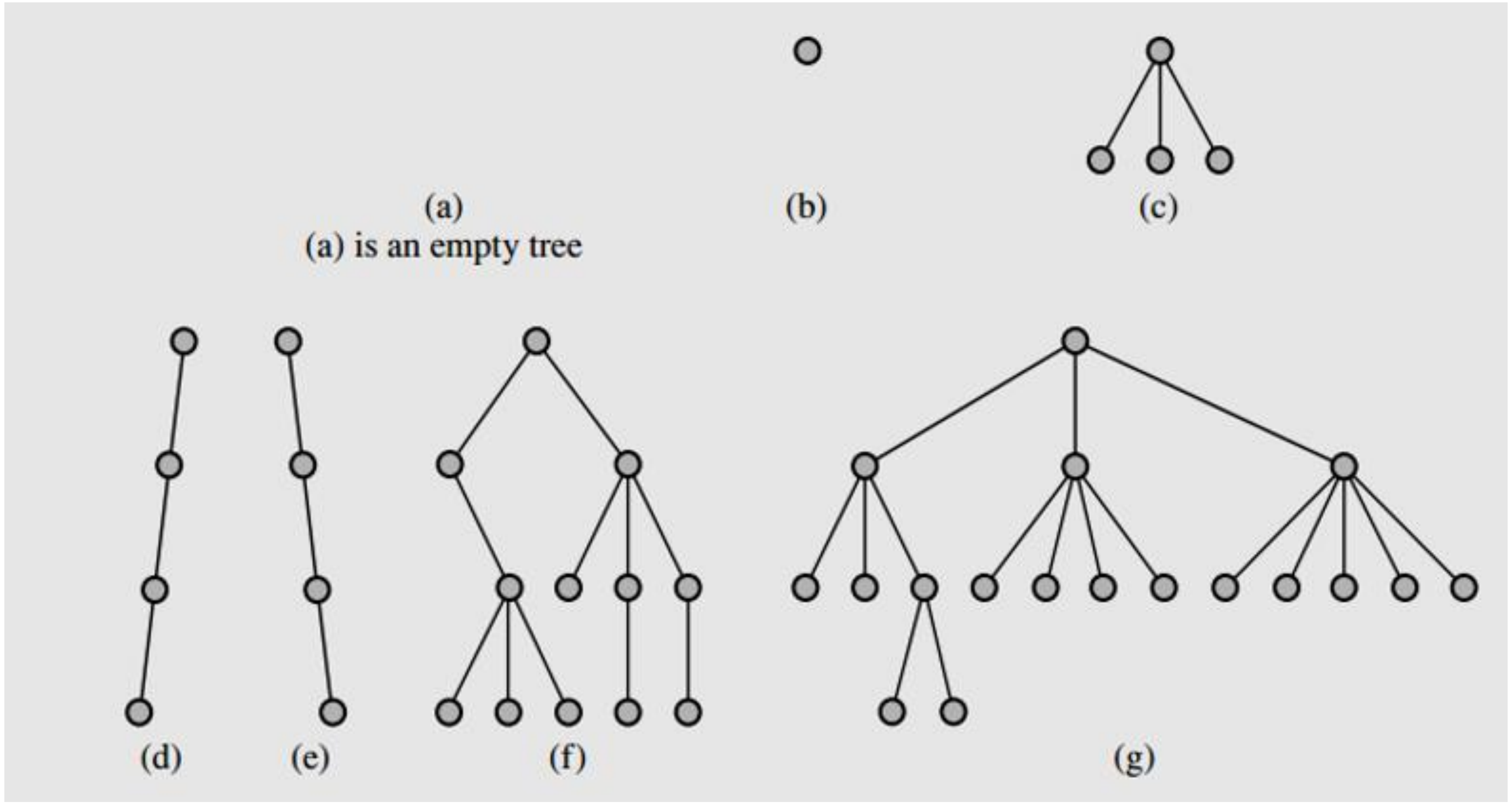
REAL WORLD TREE vs TREE DATA STRUCTURE





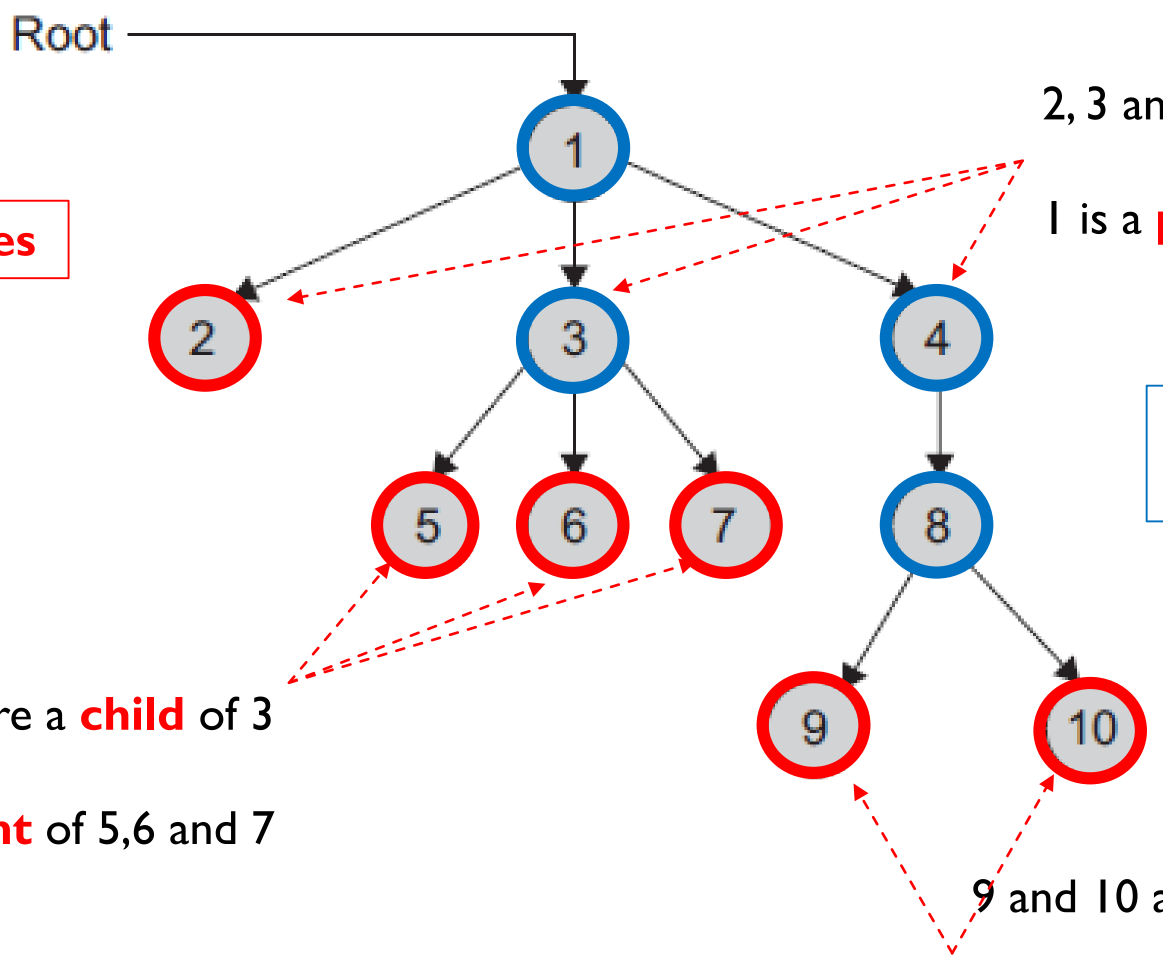
CONT..

- A tree is a collection of elements called **nodes** connected by directed (or undirected) **edges** (*that contains no cycles*). Each node contains some value.
- A tree can be empty with no nodes or a tree is a structure consisting of one node called the root and zero or one or more subtrees.
- A tree has following general properties:
  - One node is distinguished as a root;
  - Every node (exclude a root) is connected by a directed edge from exactly one other node; A direction is: parent -> children


(a)
(a) is an empty tree
(b)
(c)
(d)
(e)
(f)
(g)

6.2. BASIC TERMINOLOGIES IN TREE

- **Root**: - The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.(Root – node with no parent)
- **Child**: - The node below a given node connected by its edge downward is called its child node (left child or right child).
- **Edge:** - The connection between one node and another.
- **Parent**: - Any node except the root node has one edge upward to a node called parent.
- **Leaves**:- Nodes with no children are called leaves, or external nodes.
- **Internal nodes**:- Nodes which are not leaves, are called internal nodes. Internal nodes have at least one child.



2, 5,6,7,9 and 10 are **leaves**

2, 3 and 4 are a **child** of 1

1 is a **parent** of 2,3 and 4

1,3,4 and 8 are **internal nodes**

5, 6 and 7 are a **child** of 3

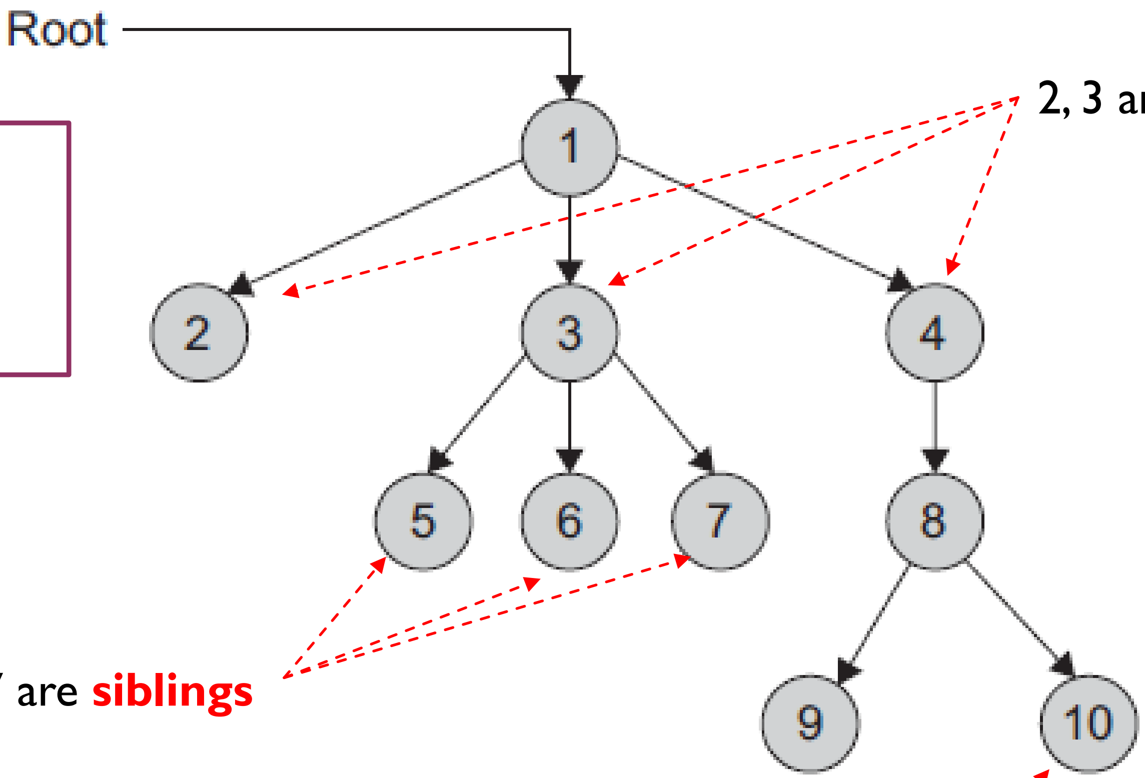
3 is a **parent** of 5,6 and 7

9 and 10 are a **child** of 8

8 is a **parent** of 9 and 10

CONT..

- **Siblings**:- Nodes with the same parent are called siblings.
- **Depth of a node**:- The depth of a node is the number of edges from the root to the node.
- **The height of a node**:- is the number of edges from the node to the deepest leaf.
- **The height of a tree**:- is a height of a root. Or (the maximum depth of any node within the tree)
- **Descendant**: - A node reachable by repeated proceeding from parent to child.(child, grandchild, grand-grandchild, etc)
- **Ancestor**: -A node reachable by repeated proceeding from child to parent.(parent, grandparent, grand-grandparent, etc.)



2, 3 and 4 are **siblings**

Height of 8 = 1
Height of 2 = 0
Height of 4 = 2
Height of the Tree = 3

Depth of 8 = 2
Depth of 2 = 1
Depth of 9 = 3

5, 6 and 7 are **siblings**

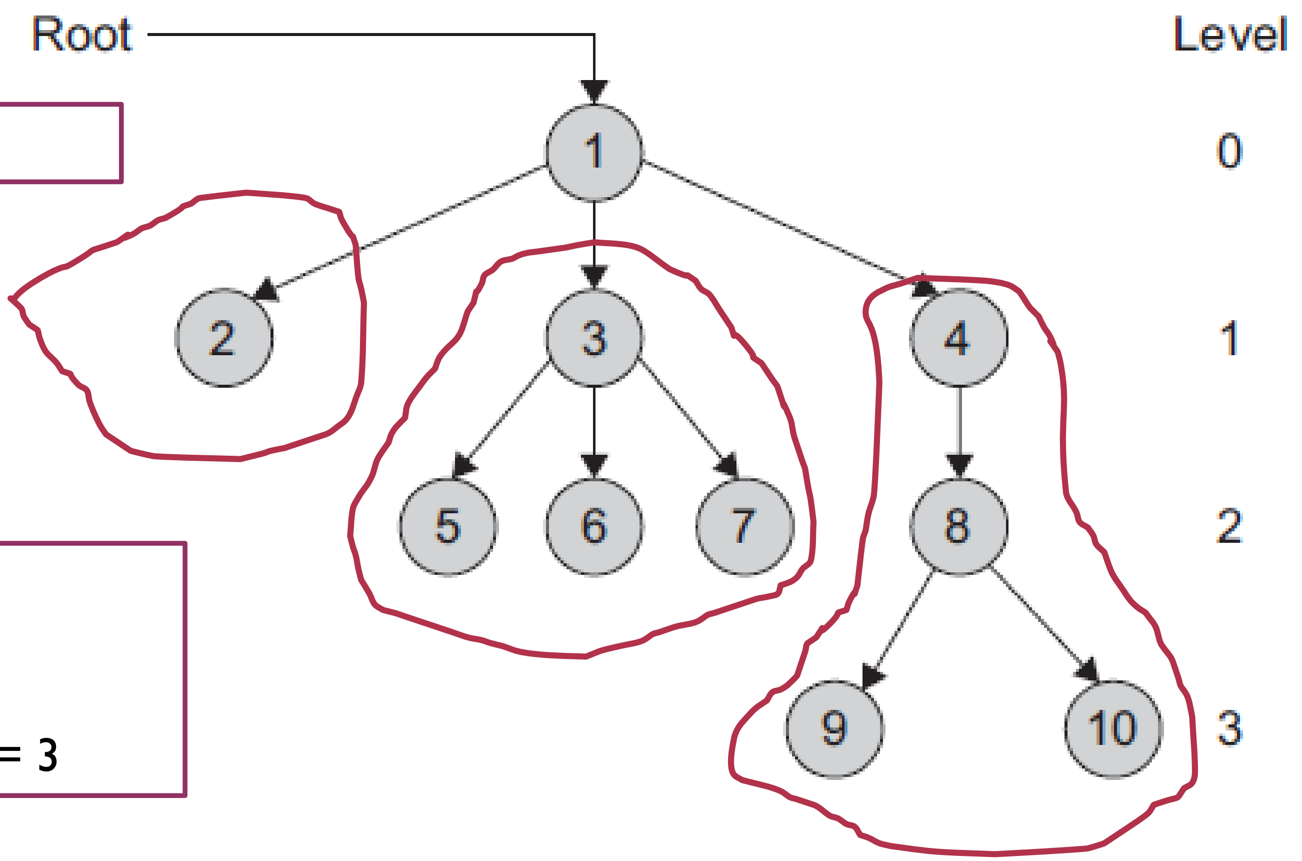
9 and 10 are **siblings**

Descendant of 4 = 8,9 and 10
Ancestor of 7 = 3 and 1

CONT..

- **Degree of a node**: is the total number of its children
- **Degree of a tree**: the maximum degree of a node in the tree.
- **Path**: -A sequence of nodes and edges connecting a node with a descendant.
- **Length of this path**: is the number of edges connecting the nodes in the path.
- **Level**: -Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **Subtree** − Subtree represents the descendants of a node.
- **Traversing** − Traversing means passing through nodes in a specific order.

Node 1 has 3 subtrees



Degree of 8 = 2
Degree of 3 = 3
Degree of 4 = 1
Degree of the Tree = 3

TYPES OF TREE

Types of Tree

- General tree, Binary Tree, Binary search tree, n-ary tree, AVL tree or height balanced binary tree, Red-Black tree…..etc.

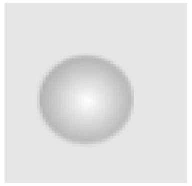
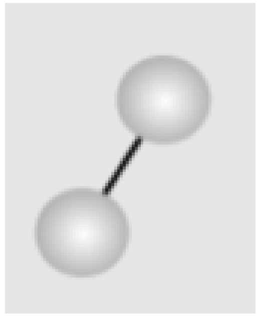
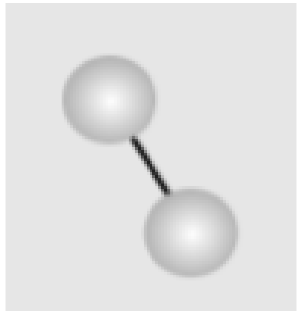
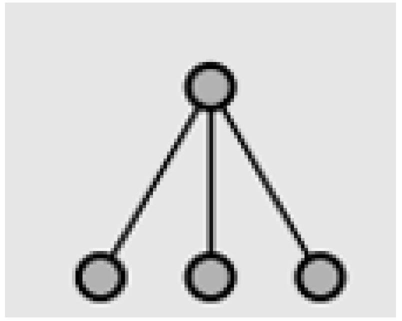
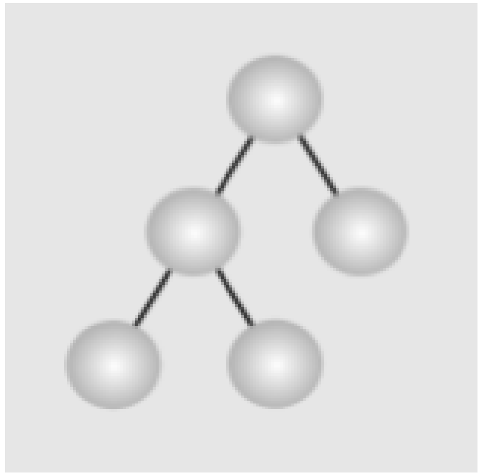
General Tree or Tree

- Every node can have any number of sub-trees, there is no maximum. Different number is possible of each node.

Binary Tree

- The most basic form of tree structure.
- A binary tree is a tree where **each node is restricted to having at most two children**, and each child is designated as either a **left child** or **right child.**

Which one of the following is Binary Tree?

A, B, C, D, E,

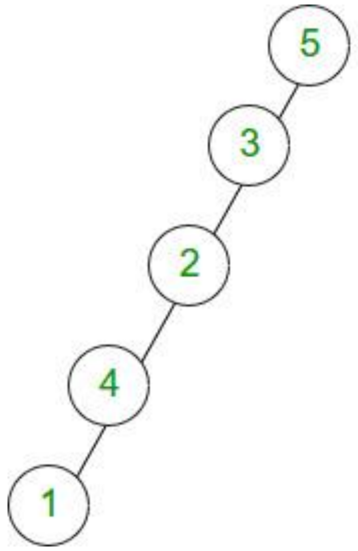
BINARY TREE

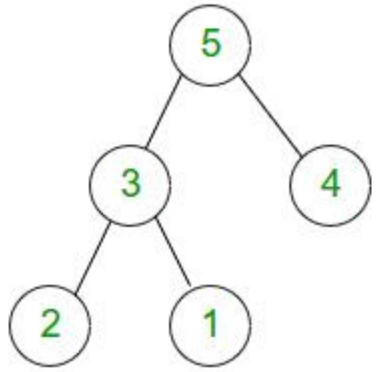
Calculating minimum and maximum heights from the number of nodes

If there are n nodes in binary tree,

- Maximum height of the binary tree is **n-1** and
- Minimum height is **floor($\log_2 n$)**

- Example; find the maximum and minimum height for a binary tree with 5 nodes?
  - Maximum height 5-1 = 4
  - Minimum height is floor($\log_2 5$) = 2





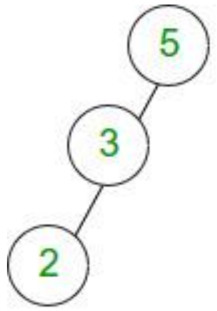
CONT..

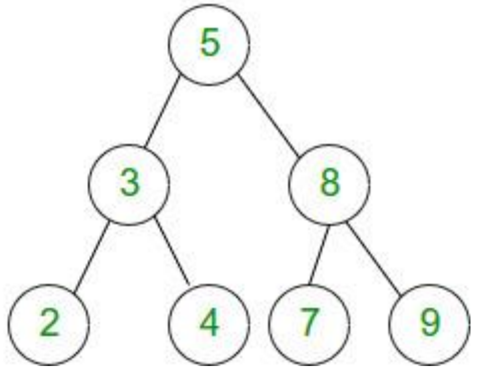
Calculating minimum and maximum number of nodes from height

If the binary tree has height h,

- Minimum number of nodes is **h+1**
- Maximum number of nodes is $\mathbf{2^{(h+1)}-1}$

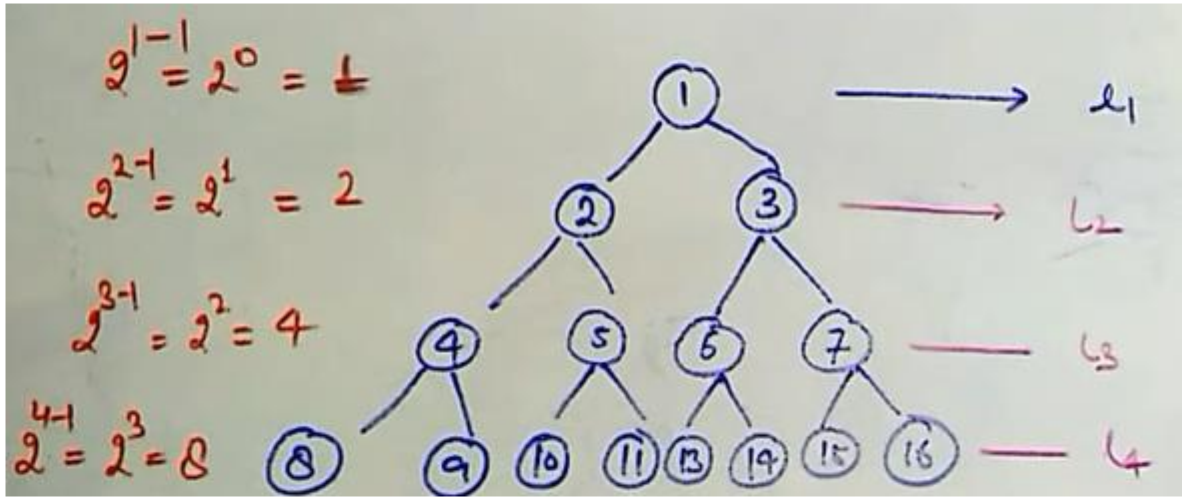
- Example; find the maximum and minimum possible number of nodes for a binary tree with Height = 2 ?
  - Maximum height 2+1 = 3
  - Minimum height $2^{(h+1)}-1$ = 7





CONT..

- In a binary tree, a maximum possible number of nodes in each level is equals to
  - 2^L (considering the root at level 0)
  - 2^{L-1} (considering the root at level 1) where L is level of the binary tree.

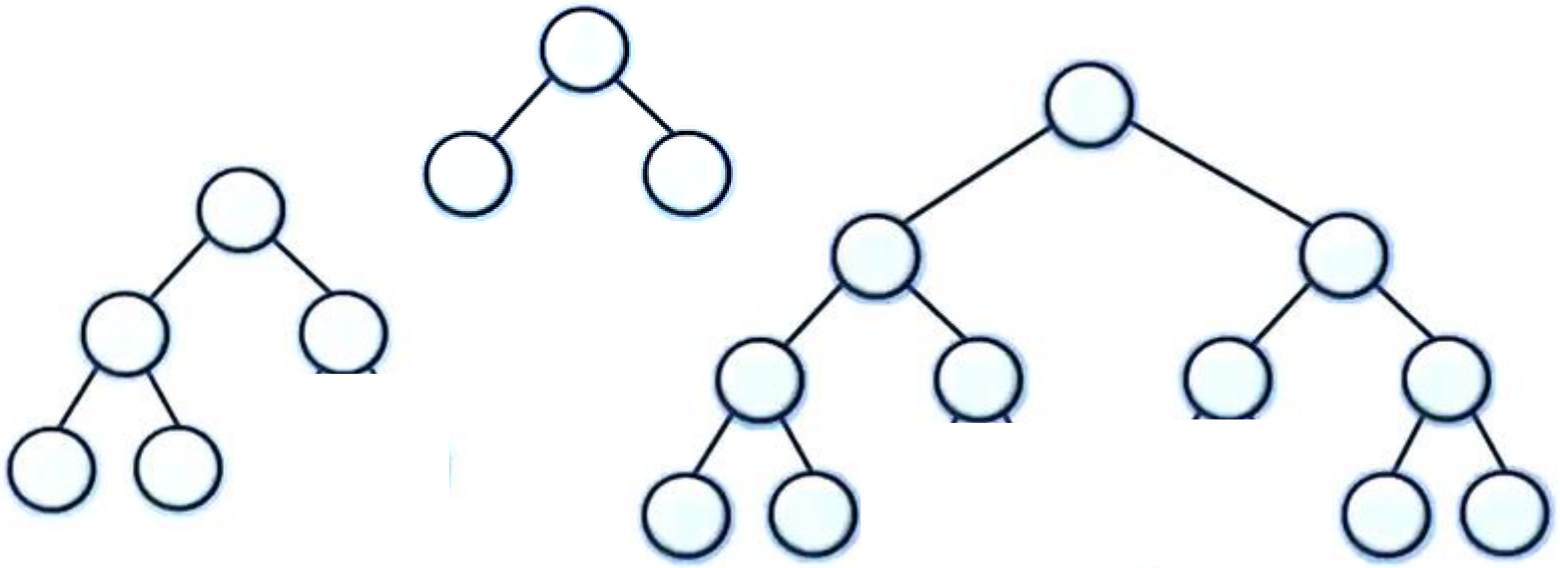


FULL BINARY TREE

- A full binary tree is a binary tree in which every tree node other than the leaves has two children.
- In other words, in a full binary tree , every node has either zero or two children.
- The total number of nodes in a full binary tree of i internal node is 2i+1 if i>=0 but 0 if the tree is empty.

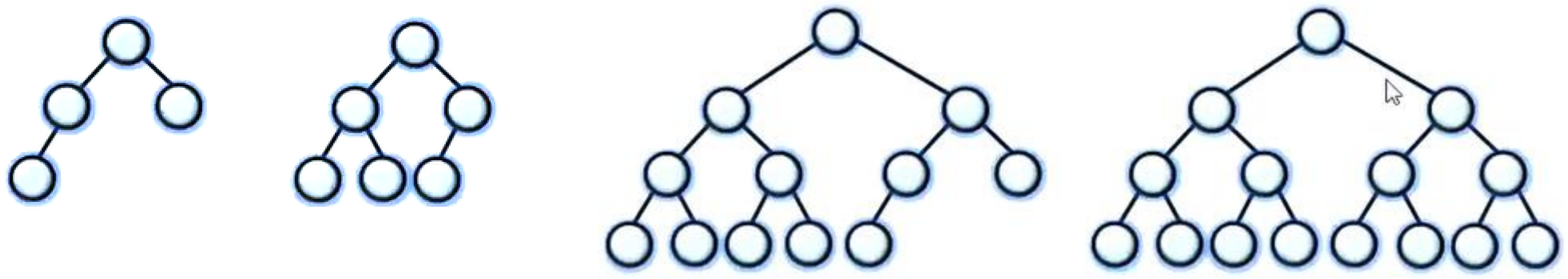
Total number of leaves i+1 Total number of internal nodes is (n-1)/2

Internal Node	Total No of Nodes
0	1 or 0
1	3
2	5
3	7



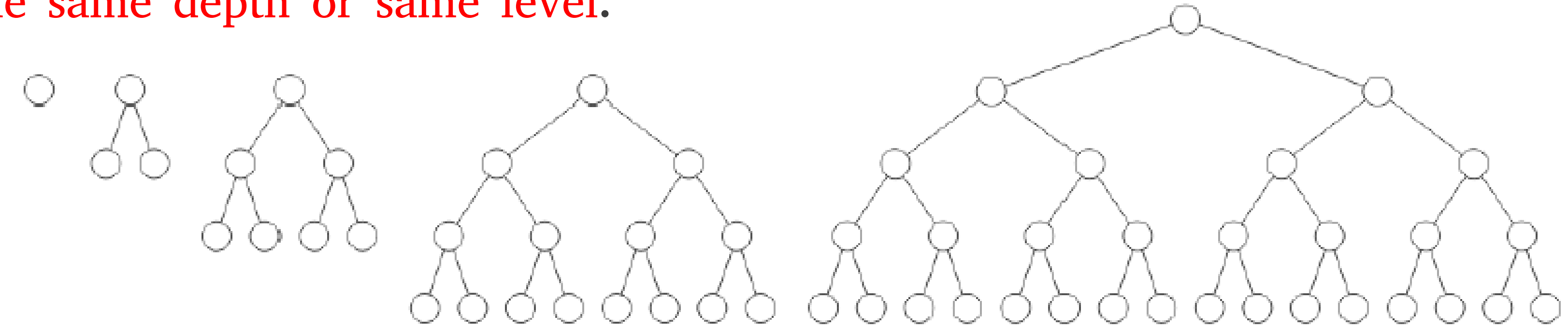
COMPLETE BINARY TREE

- A binary tree is said to be a complete binary tree if all its levels except the last level have the maximum number of possible nodes, and all the nodes of the last level appear as far left as possible.
- In a complete binary tree, all the leaf nodes are at the last and the second last level, and the levels are filled from left to right.



PERFECT BINARY TREE

- A perfect binary tree is a binary tree in which all internal nodes have two children and all leaves have the same depth or same level.



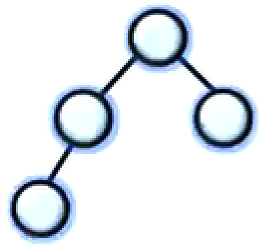
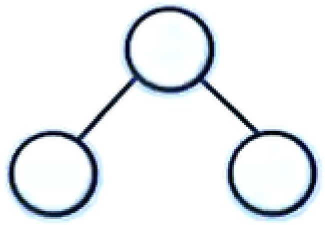
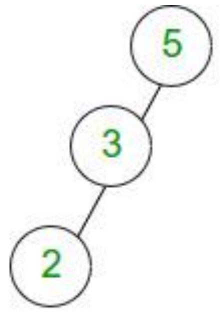
Perfect binary trees of height *h* = 0, 1, 2, 3, and 4.

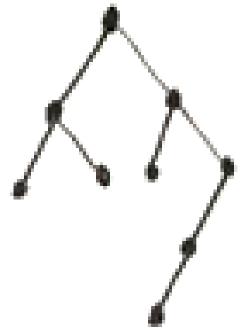
- A perfect binary tree of height h has $2^{h+1} - 1$ nodes.
- There are 2^h Leaf Nodes
- Over half of all the nodes in a perfect binary tree are leaf nodes.

BALANCED BINARY TREE

- A balanced binary tree is a binary tree in which for each node
  - The left and the right sub-tree heights differ by at most one and
  - The left sub-tree is balanced and
  - The right sub-tree is balanced.

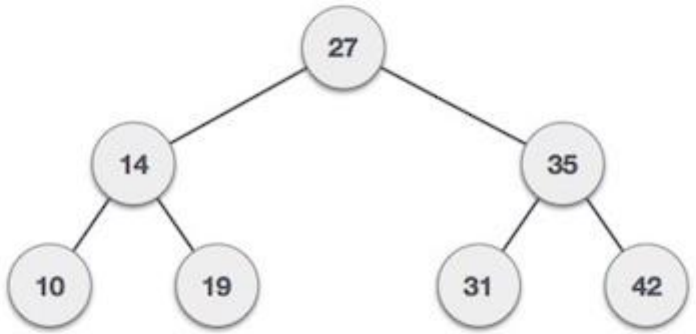
Which one of the following is balanced binary tree ?



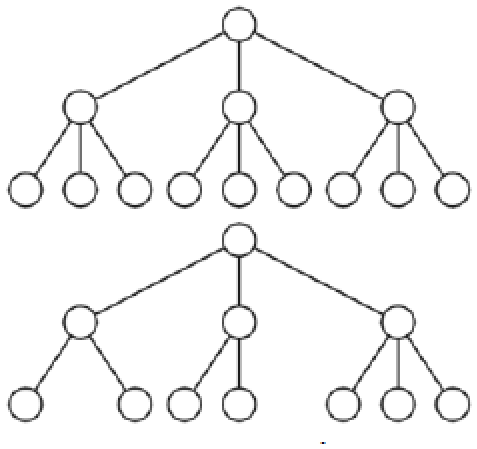
BINARY SEARCH TREE (BST)

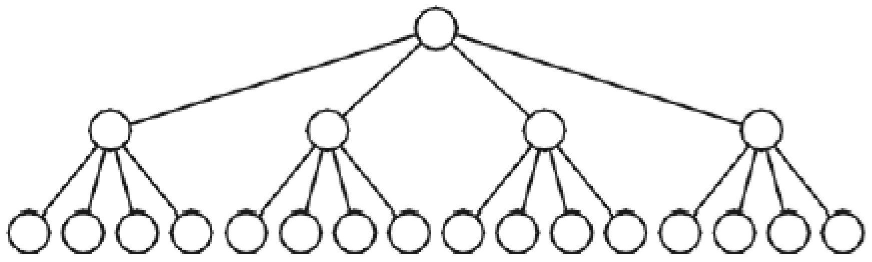
- BST is a tree in which all the nodes follow the below-mentioned properties –
  - The left sub-tree of a node has a key less than or equal to its parent node's key.
  - The right sub-tree of a node has a key greater than or equal to its parent node's key.



N-ARY TREES

- A binary tree restricts the number of children of each node to two. A more general N-ary tree restricts the number of children to N.
- An N-ary tree is a tree where each node has at most N children.
- For example, a 3-ary tree or ternary tree restricts each node to having at most three children. A quaternary tree limits its children to four.





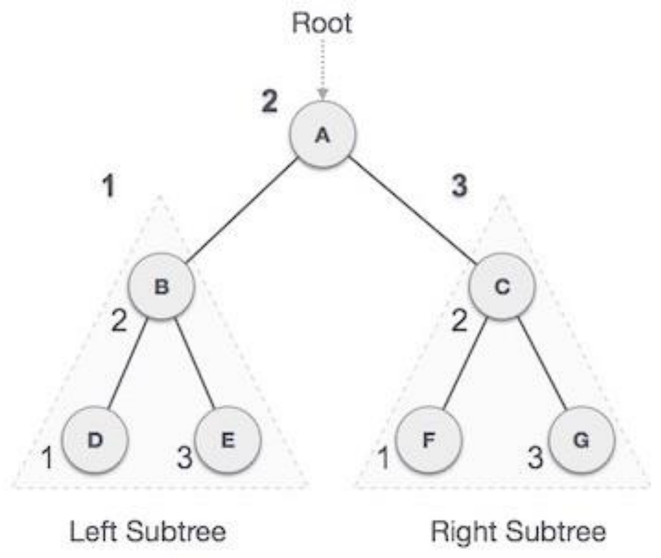
2	3	4	5	6	7	8	9	10	12
binary	ternary	quaternary	quinary	senary	septenary	octal	nonary	decimal	duodecimal

TREE TRAVERSAL

- Traversal is a process to visit all the nodes of a tree and may print their values too.
- There are three ways which we use to traverse a tree –
  - In-order Traversal
  - Pre-order Traversal
  - Post-order Traversal
- Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

IN-ORDER TRAVERSAL

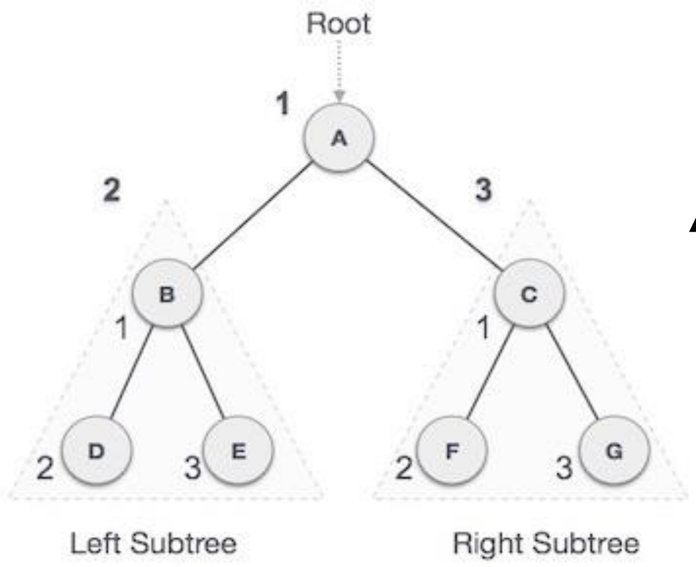
- In this traversal method, the left subtree is visited first, then the root and later the right sub-tree.
- If a binary search tree is traversed in-order, the output will produce sorted key values in an ascending order.



$D \rightarrow B \rightarrow E \rightarrow A \rightarrow F \rightarrow C \rightarrow G$

PRE-ORDER TRAVERSAL

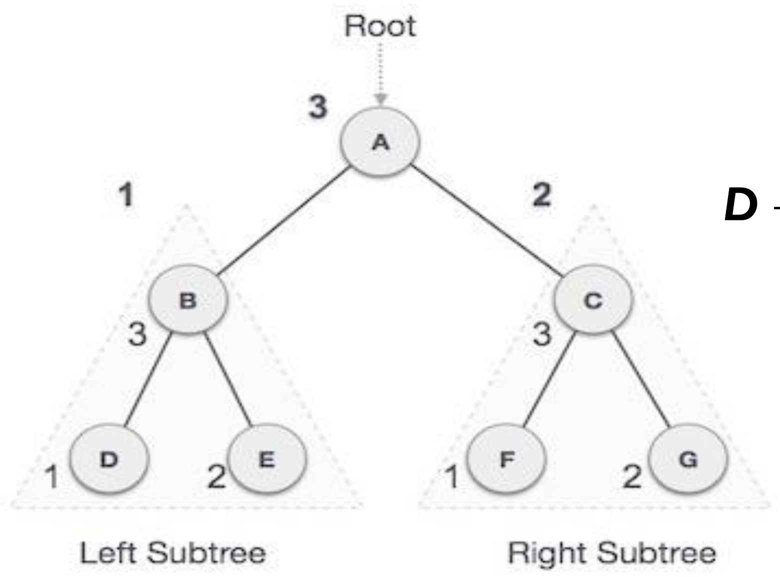
- In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



$A \rightarrow B \rightarrow D \rightarrow E \rightarrow C \rightarrow F \rightarrow G$

POST-ORDER TRAVERSAL

- In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



$D \rightarrow E \rightarrow B \rightarrow F \rightarrow G \rightarrow C \rightarrow A$

PART 2:

TREE DS

- BST Operations
- BST Implementation
  - Using array
  - Using linked list
- AVL Tree
- Heap Data Structure
  - Heap Operations
- Expression tree

OPERATIONS ON BINARY SEARCH TREE

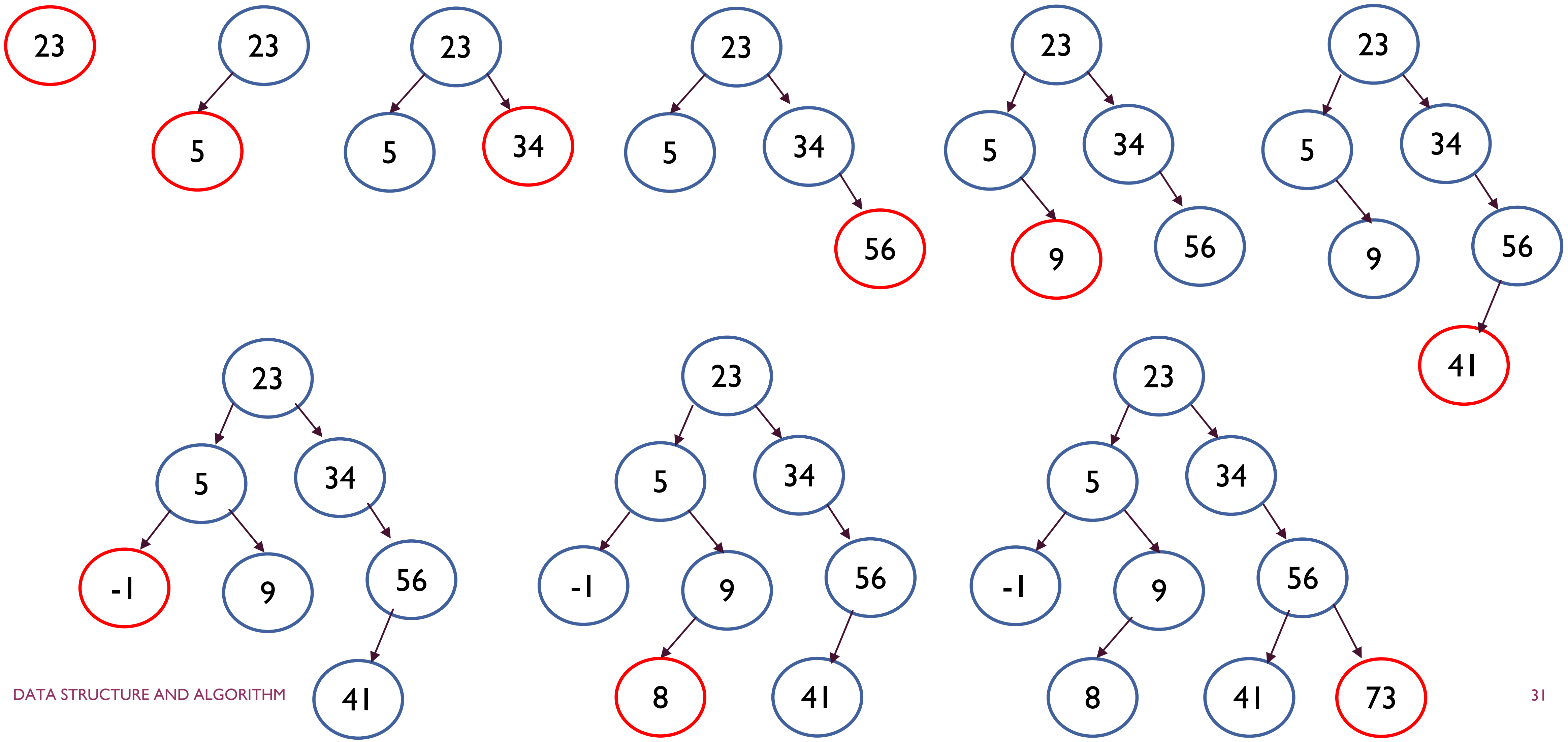
The following are the operations commonly performed on a BST:

1. **Inserting** a key: - Insert a new node with a given key in its correct place in BST
2. **Searching** a key: - Search a node in the BST with the given key
3. **Deleting** a key: - Remove a node with a given key from BST
4. **Traversing** the tree: - In-order, pre-order and post-order
5. **Finding maximum values**: - Go to the right most leaf starting from root
6. **Finding minimum values**: - Go to the left most leaf starting from the root

BST- INSERTION OPERATION

- If the tree is empty, the very first insertion creates the tree. Then the first key becomes the root.
- Afterwards, whenever an element is to be inserted,
  - First locate its proper location. Start searching from the root node, then
    - If the new key is less than the root's key, search for the empty location in the left subtree and insert the new key.
    - Otherwise, search for the empty location in the right subtree and insert the data.
- Remember no matter the value of the new key to be inserted, it will be inserted as leaf node of the tree. (unless the new key is already in the BST).

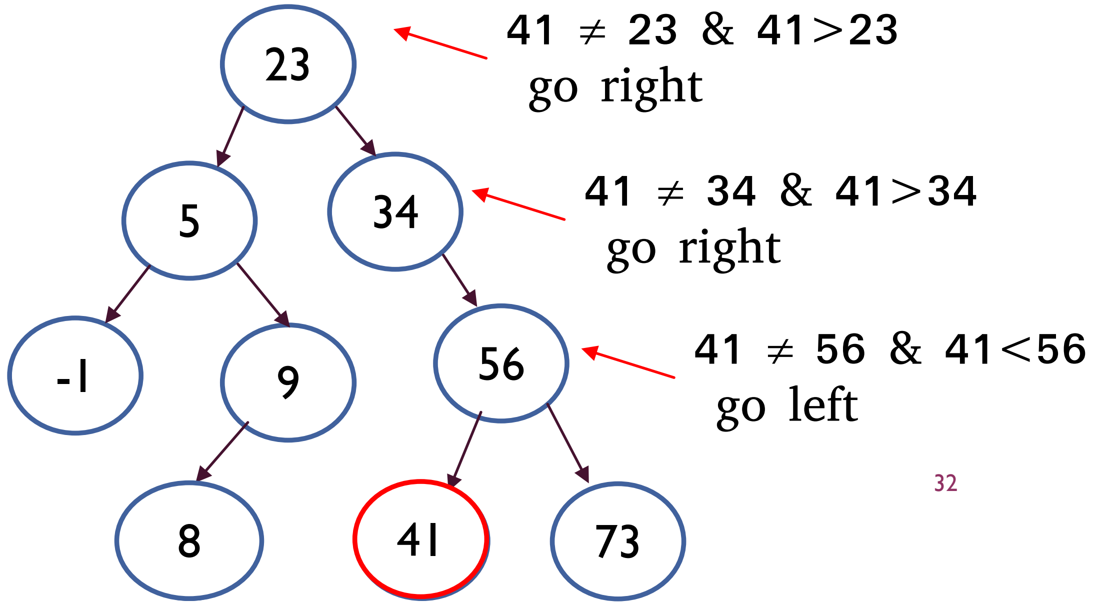
Example: Insert the following key's in BST. **23, 5, 34, 56, 9, 41, -1, 8, 73**



BST- SEARCHING OPERATION

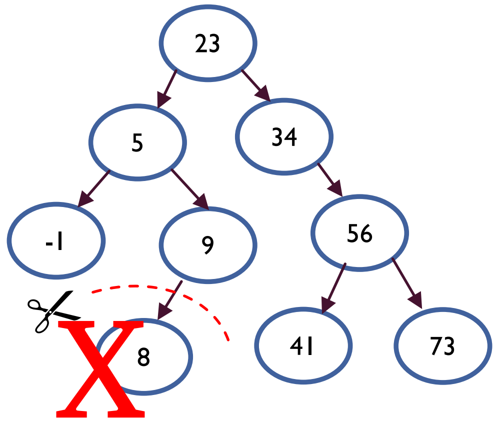
To search for a target key,

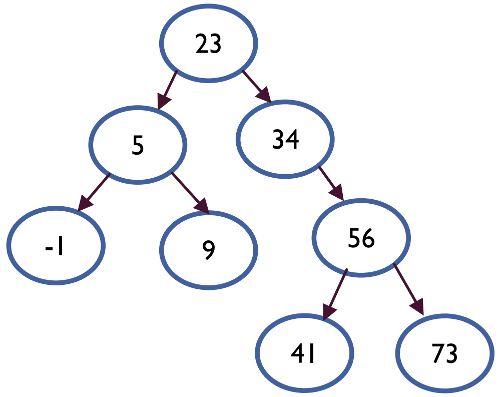
- We first compare it with the key at the root of the tree.
  - If it is the same, then the root will be returned.
  - If it is less than the key at the root, search for the target key in the left subtree,
  - else search in the right subtree.
- Example: Search for 41?



BST- DELETION OPERATION

- Complexity of deletion depends on the node
- There are three cases to consider:
  - **Case 1**: The node to be deleted is a leaf (has no children).
  - **Case 2**: The node to be deleted has one child.
  - **Case 3**: The node to be deleted has two children.
- Case 1: The node to be deleted is a leaf (has no children).
  - Simply remove the leaf node from the tree by removing the link with its parent.
- Example: Delete 8 from BST.

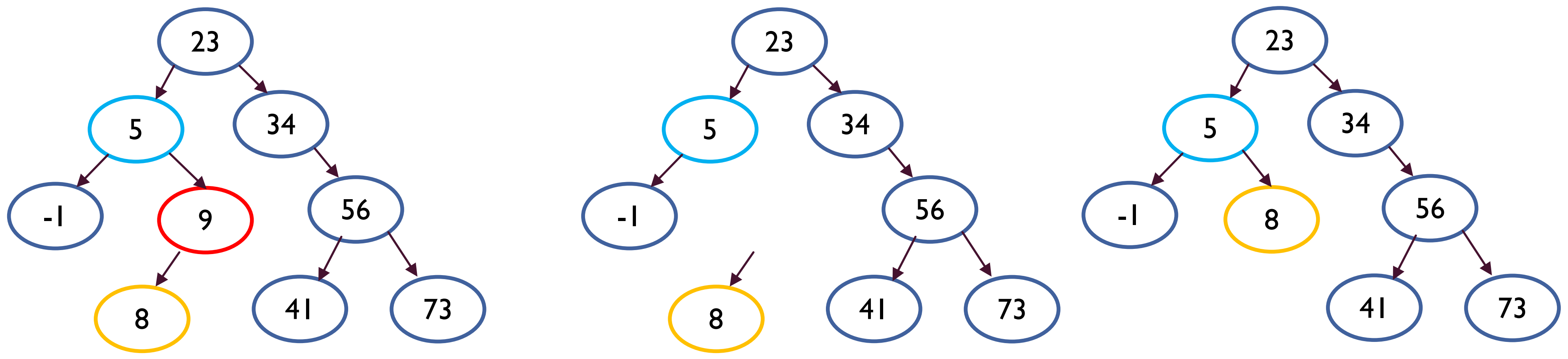




CONT..

Case 2: The node to be deleted has one child.

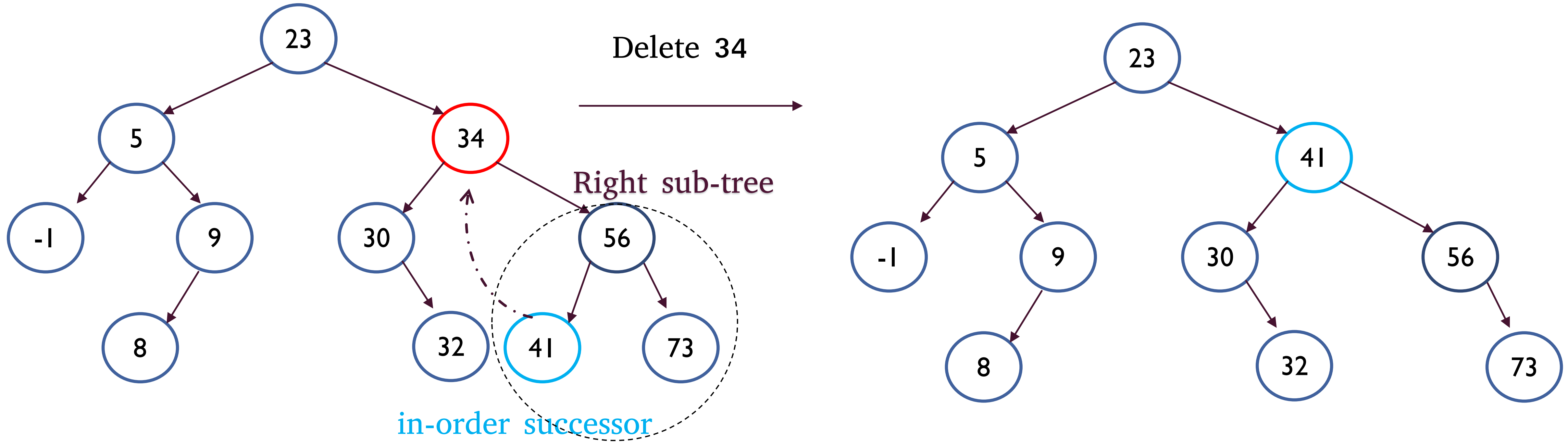
- Just make the child of the deleting node, the child of the grandparent.
- Example Delete 9 from BST

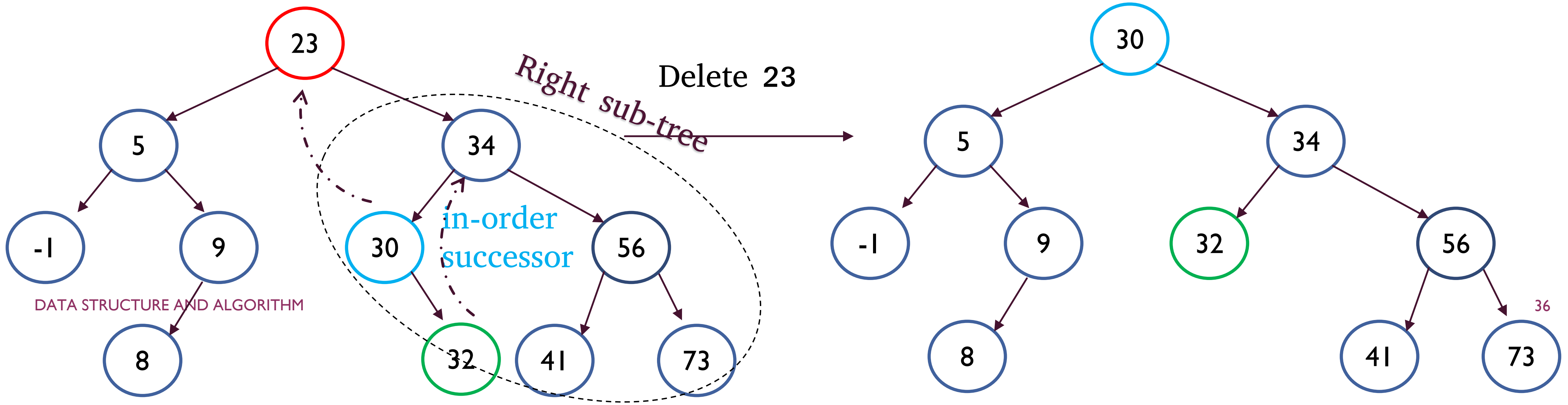


CONT..

Case 3: The node to be deleted has two child.

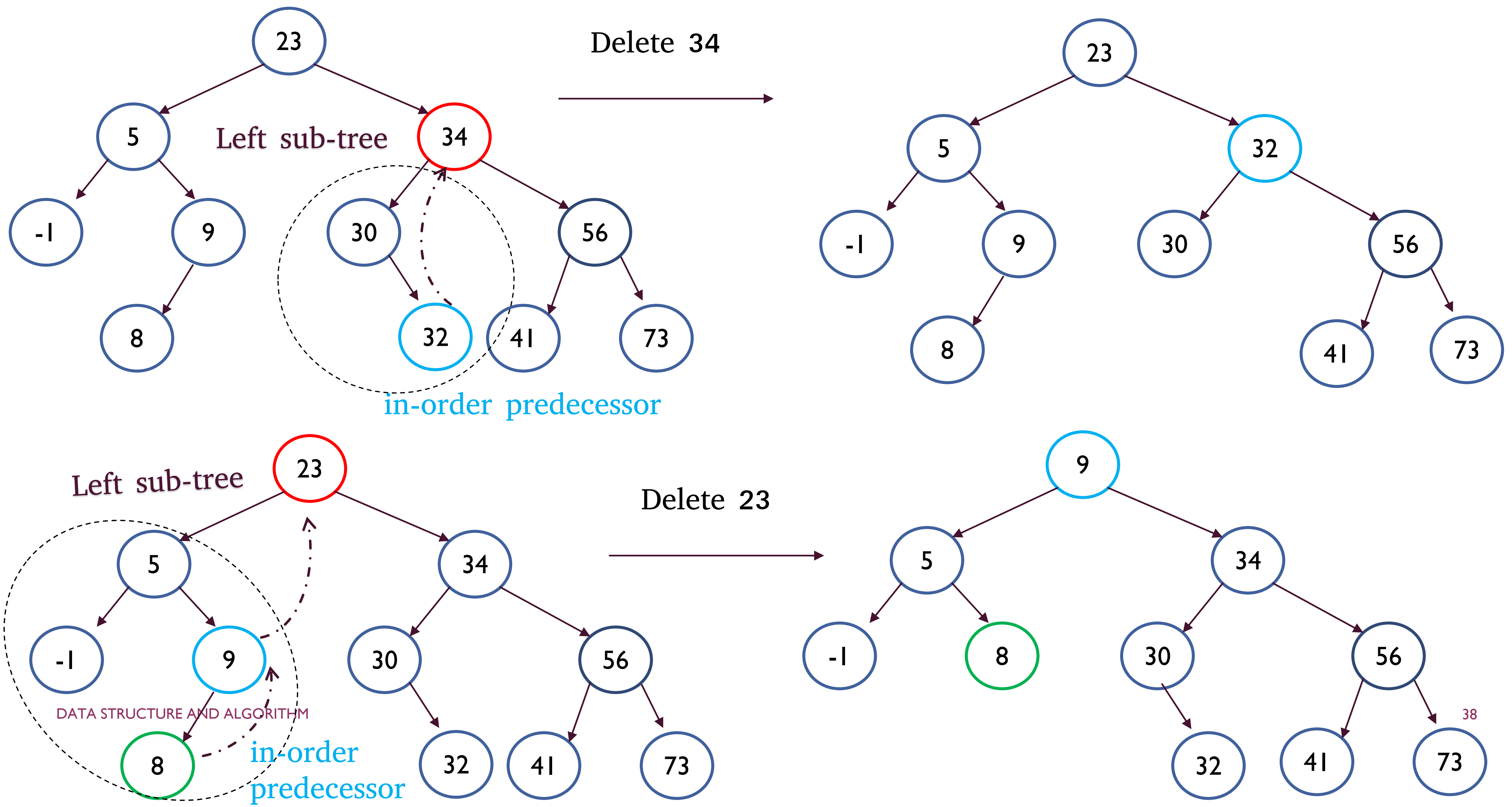
- The question is which subtrees should the parent of the deleted node be linked to, what should be done with the other subtrees, and where should the remaining subtrees be linked.
- Method 1:
  - Go to the **right subtree** of the deleting node,
  - Select the **smallest** node called in-order successor and Replace with the deleting node.
  - If the node (in-order successor node) to replace the deleting node has right child, make this node a left child of its grand parent


Delete 34
Right sub-tree
in-order successor
23
5
34
-1
9
30
56
8
32
41
73
23
5
41
-1
9
30
56
8
32
73


Right sub-tree
Delete 23
in-order successor
23
5
34
-1
9
30
56
8
32
41
73
30
5
34
-1
9
32
56
8
41
73

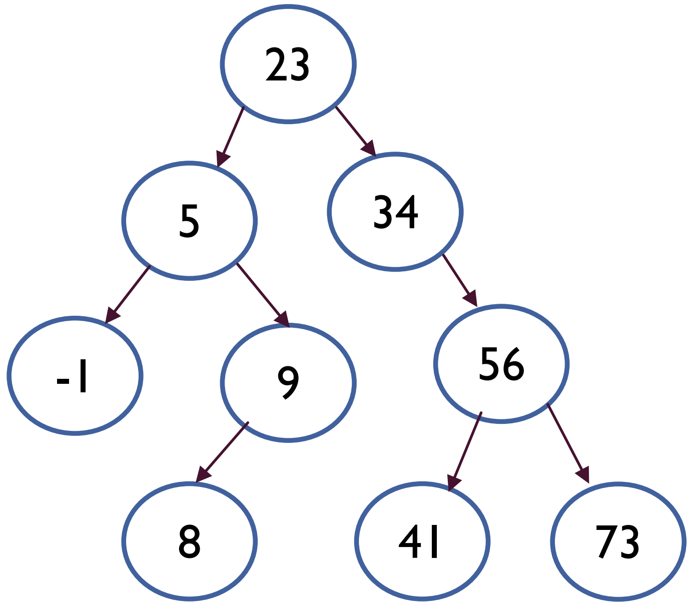
CONT..

- Method **2:**
  - Go to the **left subtree** of the deleting node,
  - Select the **greatest** node called in-order predecessor and Replace with the deleting node.
  - If the node (in-order predecessor node) to replace the deleting node has left child, make this node a right child of its grand parent


23
5
34
Left sub-tree
-1
9
30
56
8
32
41
73
in-order predecessor
Delete 34
23
5
32
-1
9
30
56
8
41
73
Left sub-tree
23
5
34
-1
9
30
56
8
32
41
73
in-order predecessor
Delete 23
9
5
34
-1
8
30
56
32
41
73

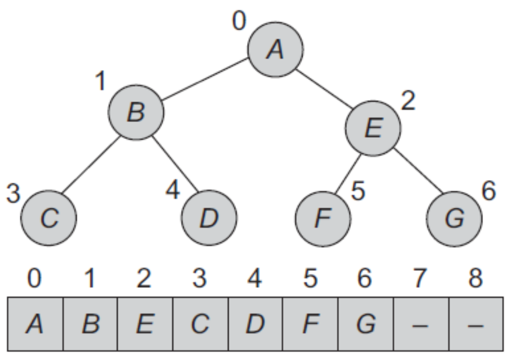
BST- OTHER OPERATIONS

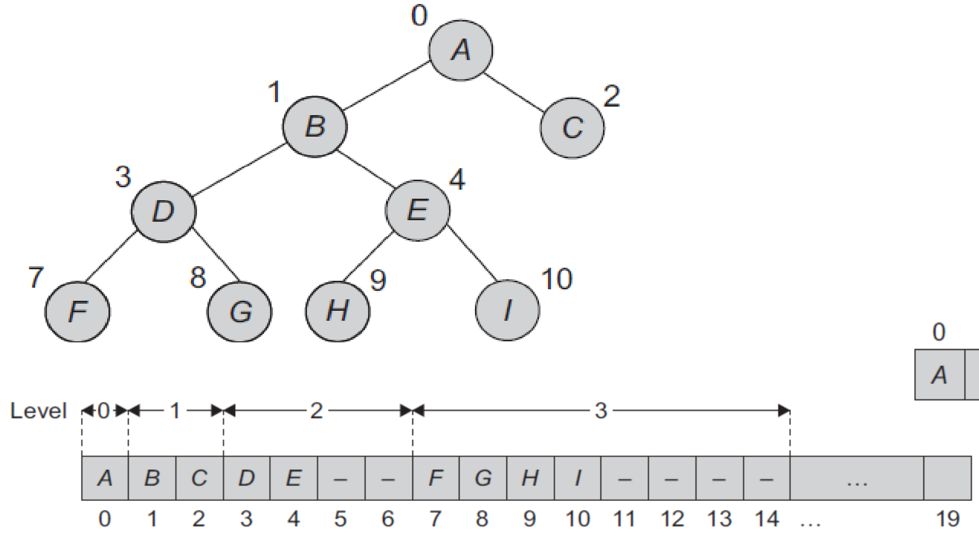
- **Traversing** the tree: -
  - In-order = **-1, 5, 8, 9, 23, 34, 41, 56, 73**
  - pre-order = **23, 5, -1, 9, 8, 34, 56, 41, 73**
  - post-order = **-1, 8, 9, 5, 41, 73, 56, 34, 23**
- **Finding maximum values**: -
  - Go to the right most leaf starting from root = **73**
- **Finding minimum values**: -
  - Go to the left most leaf starting from the root = **-1**

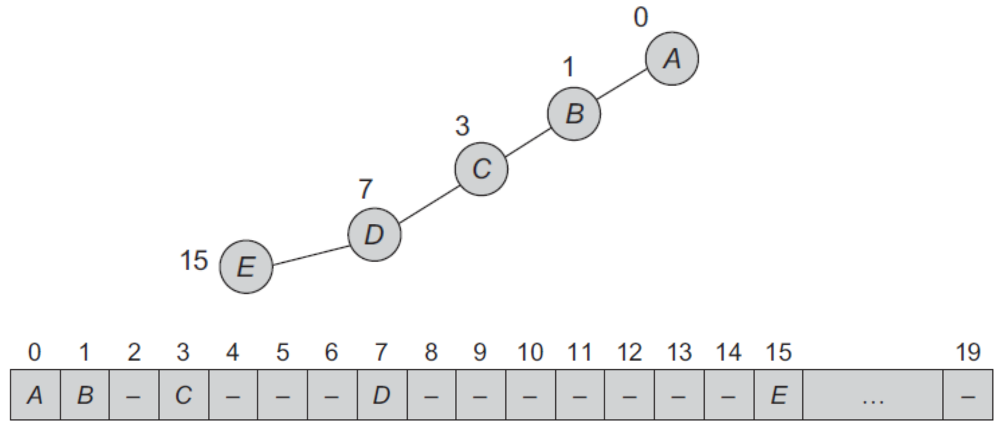


BST IMPLEMENTATION USING ARRAY

- One of the ways to represent a tree using an array is to store the nodes **level-by-level**, starting from the level 0 where the root is present.
- A complete binary tree of height h has ($2^{h+1} - 1$) nodes in it. The nodes can be stored in a one-dimensional array of size **$2^{h+1} - 1$** is needed for the same.





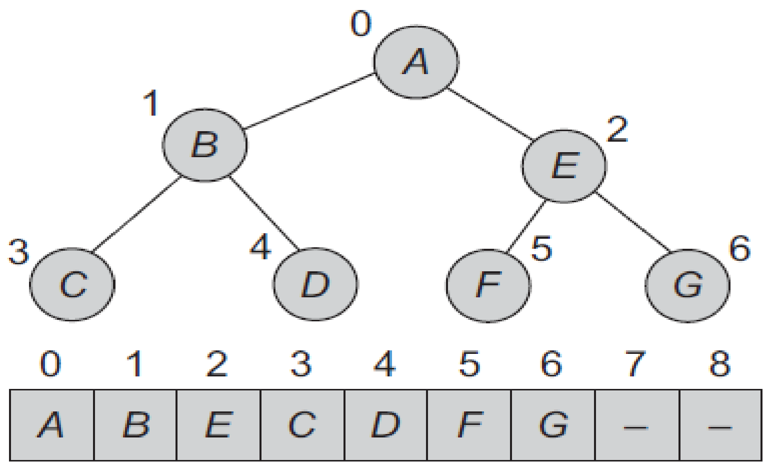


This representation of binary trees using an array seems to be the easiest. Certainly, it can be used for all binary trees.

CONT..

For any node with index i, **0 ≤ i ≤ n - 1**, we can find the location of its parent, left child and right child using the following formula.

- 1. Parent_index = floor((i - 1)/2) if i ≠ 0; if i = 0, then it is the root (has no parent).
- 2. Lchild_index = **2i + 1** if 2i + 1 ≤ n - 1; if 2i ≥ n, then i has no left child.
- 3. Rchild_index = **2i + 2** if 2i + 2 ≤ n - 1; if (2i + 1) ≥ n, then i has no right child.



0	1	2	3	4	5	6	7	8
A	B	E	C	D	F	G	–	–

ADVANTAGES AND DISADVANTAGES

Advantages:

1. Any node can be accessed from any other node by calculating the index.
2. The data is stored without any pointers to its successor or predecessor.
3. In the programming languages, where dynamic memory allocation is not possible (such as BASIC, fortran), array representation is the only means to store a tree.

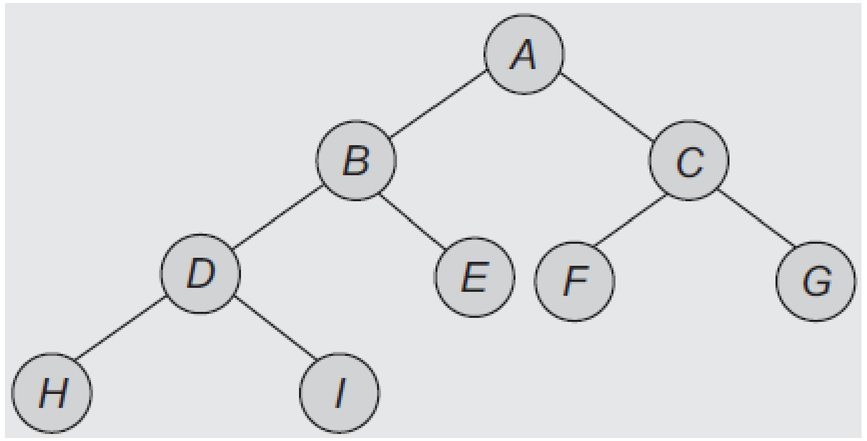
Disadvantages:

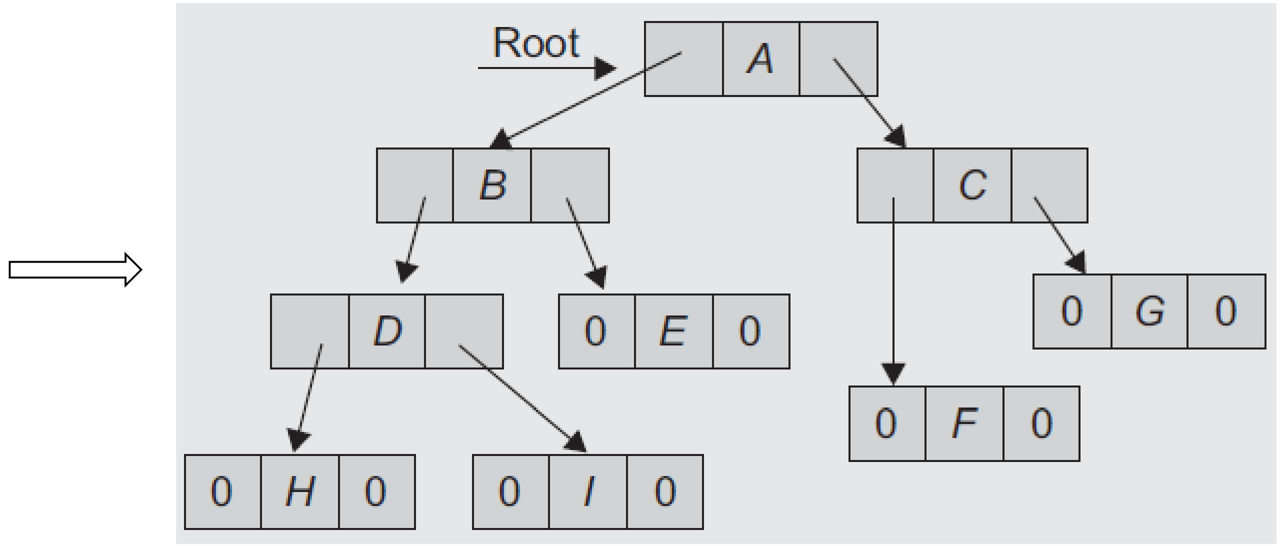
1. Other than full binary trees, majority of the array entries may be empty.
2. It allows only static representation.
3. Insertion and deletion operation requires lots of data movement.

These problems can be overcome by the use of linked representation.

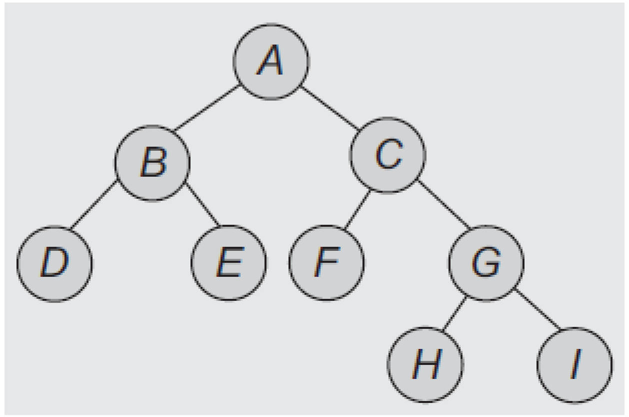
BST-LINKED LIST IMPLEMENTATION

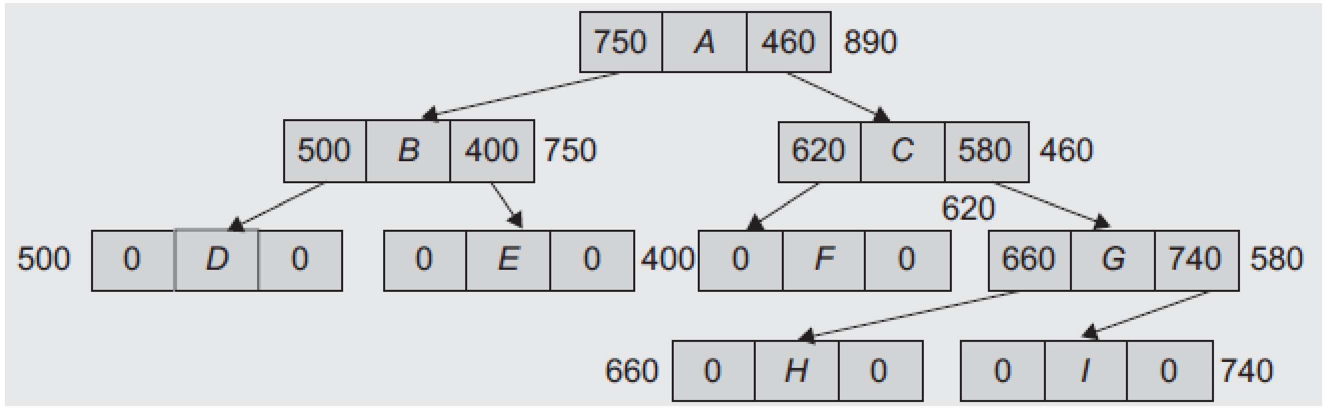
- Using linked list, each node will have three fields — Lchild, Data, and Rchild.
- If needed, the fourth parent field can be included.
- The root of the tree is stored in the data member root of the tree.
  - This data member provides an access pointer to the tree.





Here, 0 (zero) stored at Lchild or Rchild fields represents that the respective child is not present. Alternately NULL can be used instead of zero.





Address	Nodes		
	Child	Data	Rchild
500	0	*D*	0
750	500	*B*	400
400	0	*E*	0
890	750	*A*	460
620	0	*F*	0
460	620	*C*	580
660	0	*H*	0
580	660	*G*	740
740	0	*I*	0

ADVANTAGES AND DISADVANTAGES

Advantages:

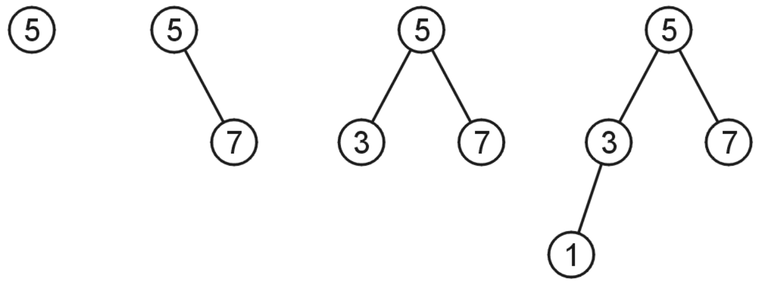
1. The drawbacks of the sequential representation are overcome in this representation. In addition, for unbalanced trees, the memory is not wasted.
2. Insertion and deletion operations are more efficient in this representation.
3. It is useful for dynamic data.

Disadvantages:

1. In this representation, there is no direct access to any node.
2. As compared to sequential representation, the memory needed per node is more. This is due to two link fields (left child and right child for binary trees) in the node.
3. The programming languages not supporting dynamic memory management would not be useful for this representation.

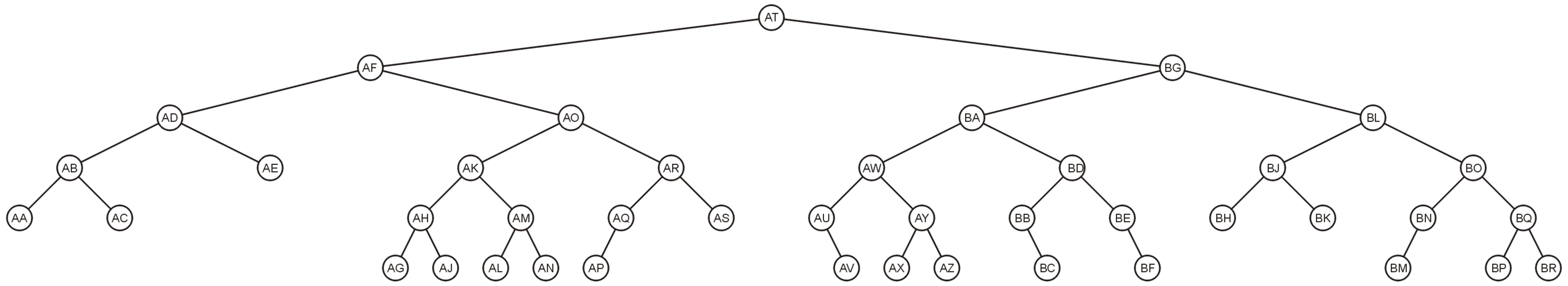
AVL TREES

- AVL tree is a self-balancing **binary search tree**.
- Named after Adelson-Velskii and Landis
- Balance is defined by comparing the height of the two sub-trees
- In AVL tree, the heights of the two child subtrees of any node differ by at most one.
- If at any time they differ by more than one, rebalancing is done to restore this property.
- Insertion and deletion may require the tree to be rebalanced by one or more tree rotation.
- Recall:
  - An empty tree has height -1
  - A tree with a single node has height 0



AVL trees with 1, 2, 3, and 4 nodes:

AVL tree or not?



WHY AVL TREE?

- Most BST operations takes **O(h)** time where h is the height of BST.
- The cost of this operations may become **O(n)** for a skewed binary tree.
- If we make sure that height of the tree remains **O(Log n)** after every insertion and deletion, then we can guarantee an upper bound of **O(Log n)** for all these operations.
- The height of AVL tree is always **O(Log n)** where n is the number of nodes in the tree.

- To keep the tree height-balanced, we have to find out the balance factor of each node in the tree after every insertion or deletion.
- The balance factor of a node T, BF(T), in a binary tree is hL - hR, where hL and hR are the heights of the left and right subtrees of T, respectively.
- For any node T in an AVL tree, the BF(T) is equal to -1, 0, or 1.
- If a node is inserted or deleted from a balanced tree, then it may become unbalanced.
- So to rebalance it, the position of some nodes can be changed in proper sequence.
- This can be achieved by performing rotations of nodes.

BALANCING AVL TREE

- Depends on the scenario whether a rotation should be performed towards left or right.
- To balance itself, an AVL tree may perform the following four kinds of rotations

1. **Left rotation** (for Right Right (RR) problem)
2. **Right rotation** (for Left Left (LL) problem)

$\left.\right\}$ Single rotations

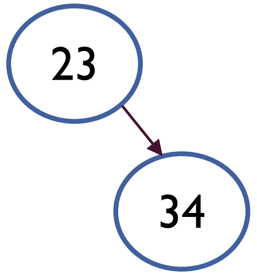
3. **Left-Right (LR) rotation** (for Left Right (LR) problem)
4. **Right-Left (RL) rotation** (for Right Left (RL) problem)

$\left.\right\}$ Double rotations

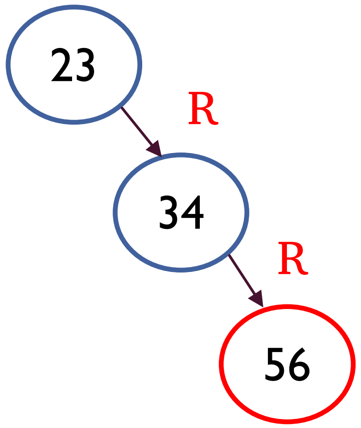
To have an unbalanced tree, we at least need a tree of height 2.

LEFT ROTATION

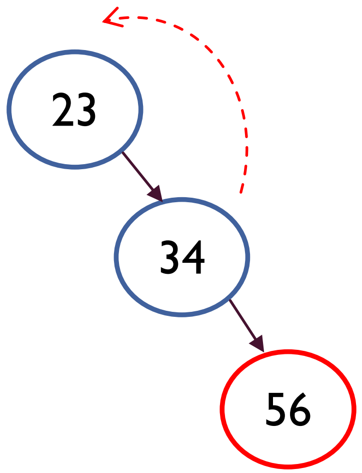
- If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



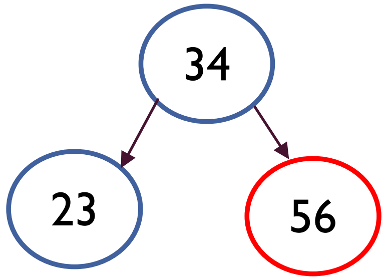
Balanced AVL



Unbalanced AVL
Node 23 has become unbalanced as 56 is inserted in the right subtree of 23's right subtree.



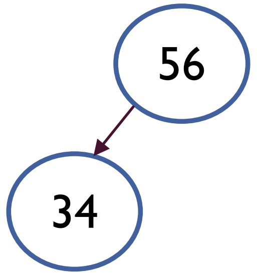
Left Rotation



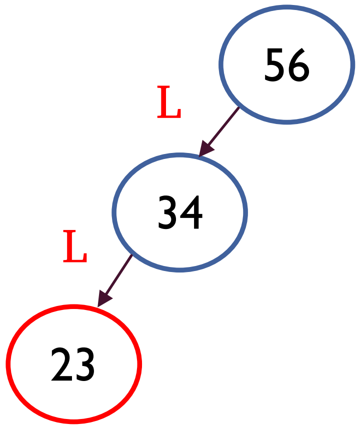
Balanced AVL

RIGHT ROTATION

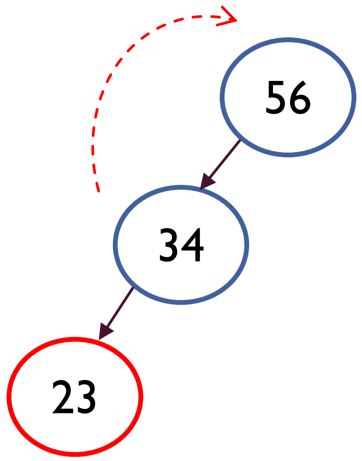
- If a tree becomes unbalanced, when a node is inserted into the left subtree of the left subtree, then we perform a single right rotation –



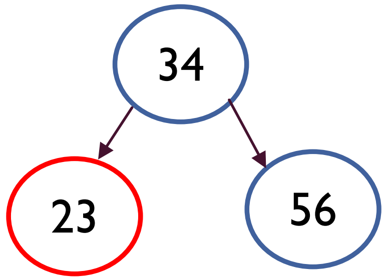
Balanced AVL



Unbalanced AVL
Node 23 has become unbalanced as 56 is inserted in the left subtree of 23's left subtree.



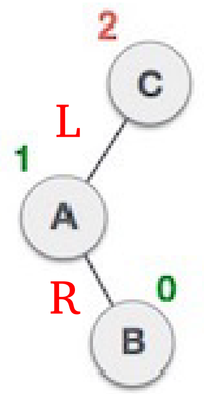
Left Rotation

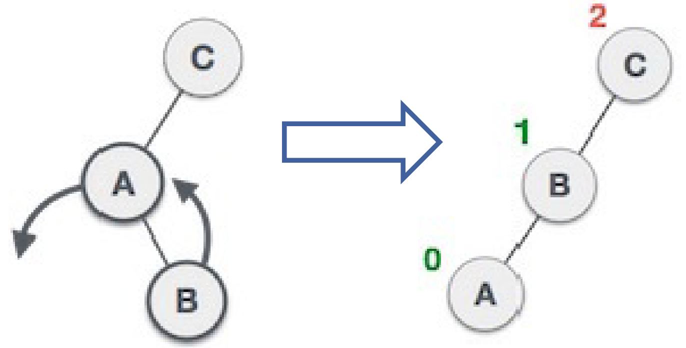


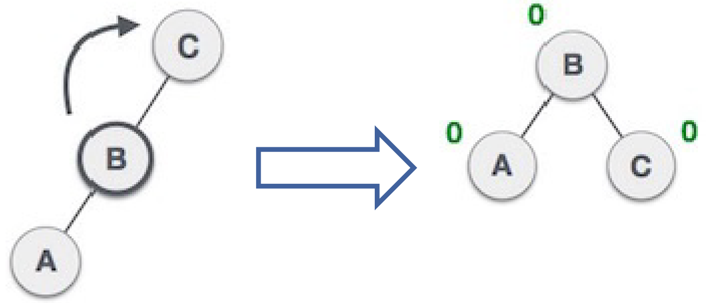
Balanced AVL

LEFT-RIGHT ROTATION

- Double rotations are slightly complex version of already explained versions of rotations.
- A left-right rotation is a combination of **left rotation followed by right rotation**.
- If a node inserted into the **right subtree of the left subtree** and the AVL becomes unbalanced we perform left-right rotation.







Unbalanced AVL

Node C has become unbalanced as node B is inserted in the right subtree of Node C's left subtree.

We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.

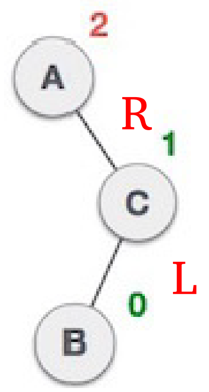
Node C is still unbalanced, however now, it is because of the left-subtree of the left-subtree.

We shall now right-rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree.

The tree is now balanced.

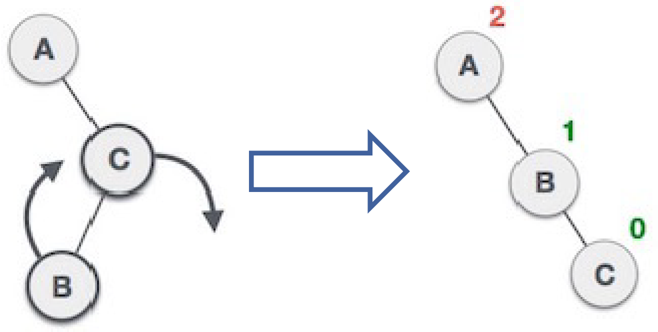
RIGHT-LEFT ROTATION

- A right-left rotation is a combination of **right rotation followed by left rotation.**
- If a node inserted into the **left subtree of the right subtree** and the AVL becomes unbalanced we perform left-right rotation.



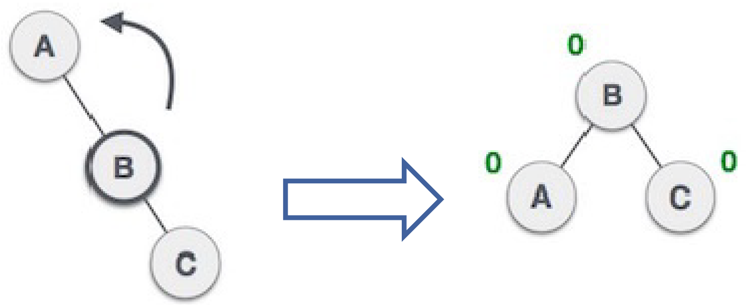
Unbalanced AVL

Node A has become unbalanced as node B is inserted in the left subtree of Node A's Right subtree.



We first perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.

Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation..

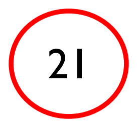


A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.

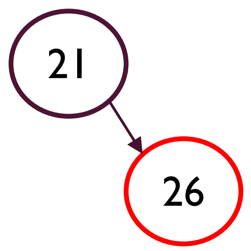
The tree is now balanced.

Create AVL Tree for the following unsorted list of keys.

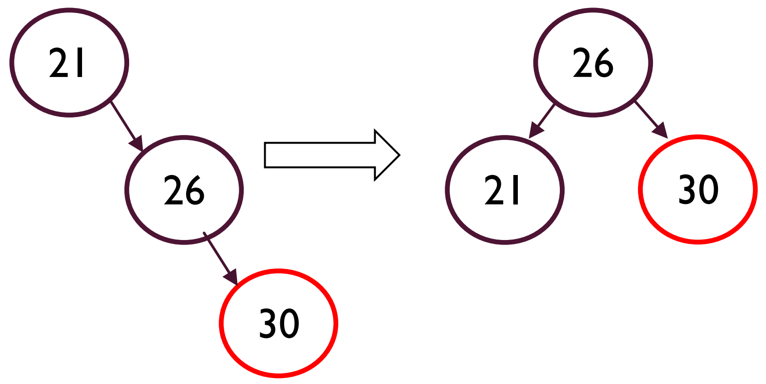
- 21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7



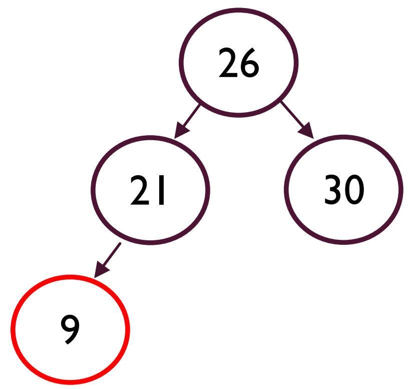
A.



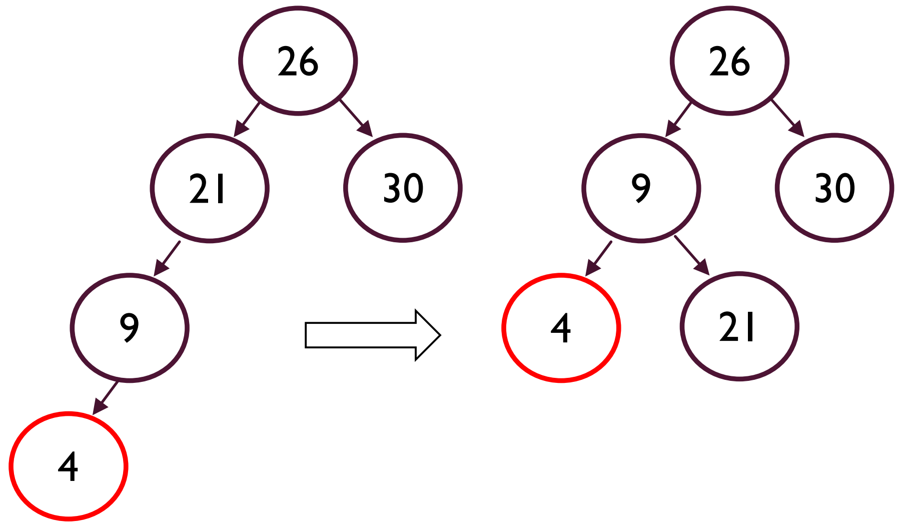
B.



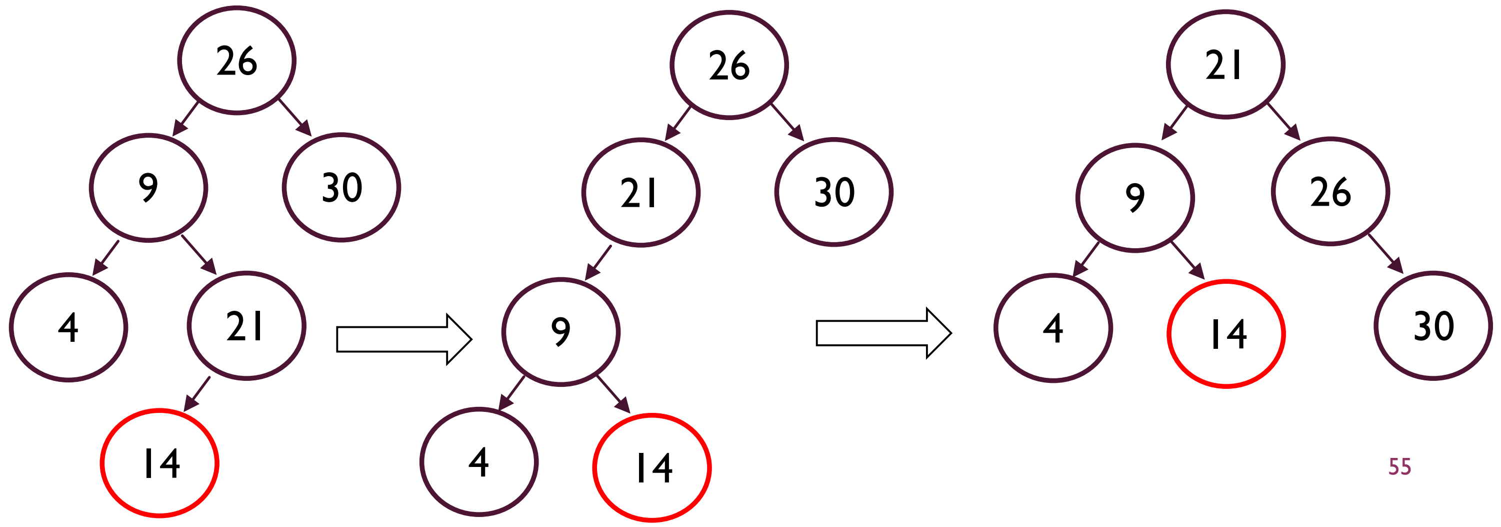
C.



D.



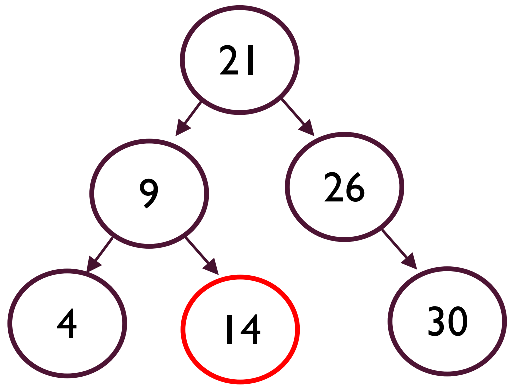
E.



F.

Create AVL Tree for the following unsorted list of keys.

- 21, 26, 30, 9, 4, 14, 28, 18, 15, 10, 2, 3, 7



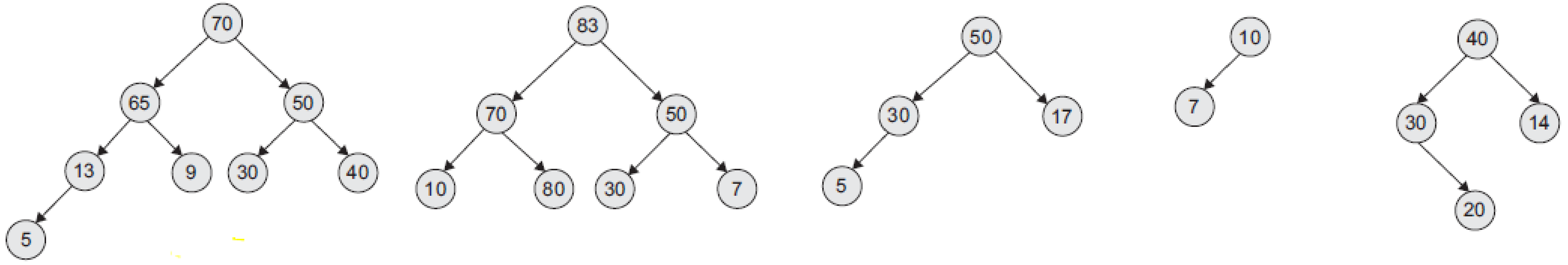
Continue inserting ………………..

HEAP DATA STRUCTURE

A heap is a binary tree having the following properties:

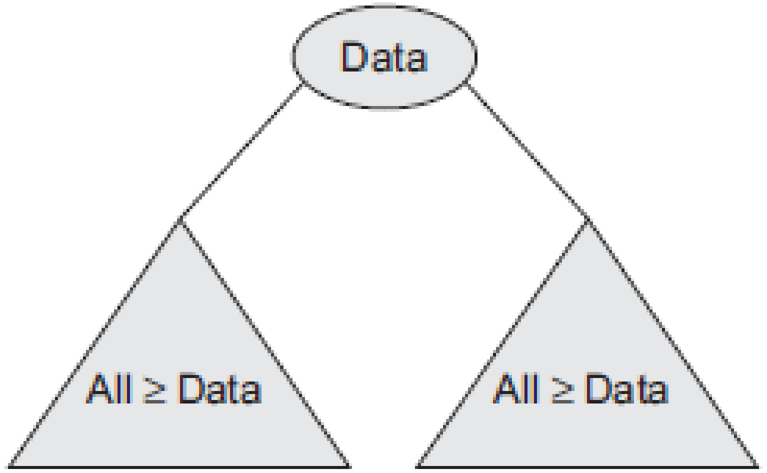
1. It is **a complete binary tree**, that is, each level of the tree is completely filled, except the bottom level, where it is filled from left to right.
2. It satisfies the heap-order property , that is,
   - The key value of each node is greater than or equal to the key value of its children, or
   - The key value of each node is lesser than or equal to the key value of its children.

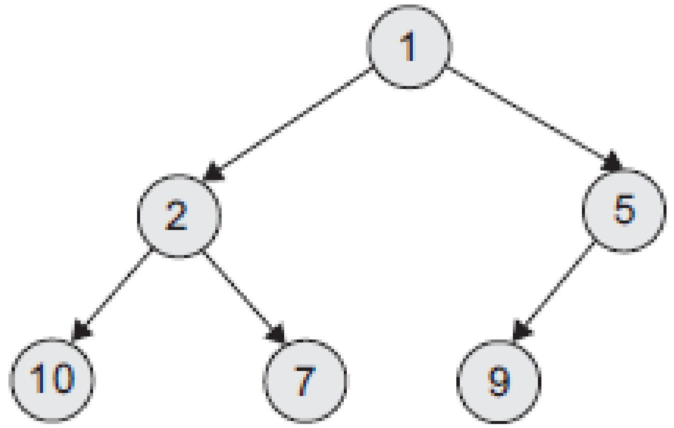
Which one of the following is Heap?



MIN-HEAP

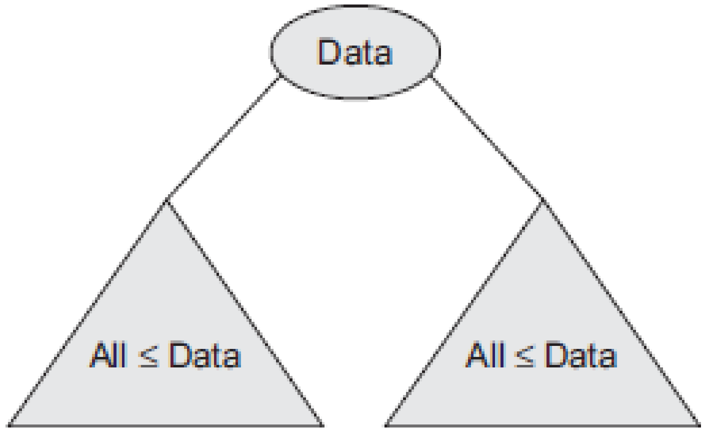
- In min-heap, the **key value of each node is lesser than or equal to the key value of its children**.
  - Parent key must <= children's key
- In addition, every path from root to leaf should be sorted in **ascending order**.

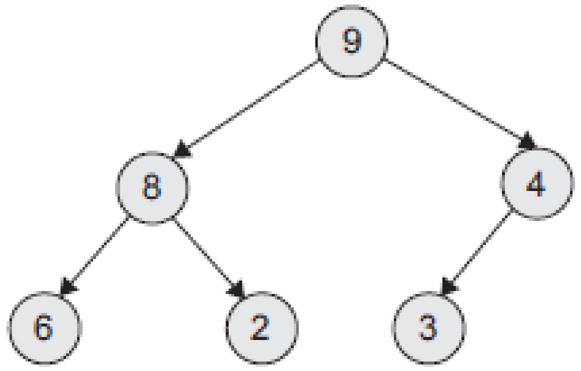




MAX-HEAP

- A max-heap is where the **key value of a node is greater than or equal to the key value of its children**.
  - Parent key must >= children's key
- In addition, every path from root to leaf should be sorted in **descending order**.
- In general, whenever the term 'heap' is used by itself, it refers to a max-heap





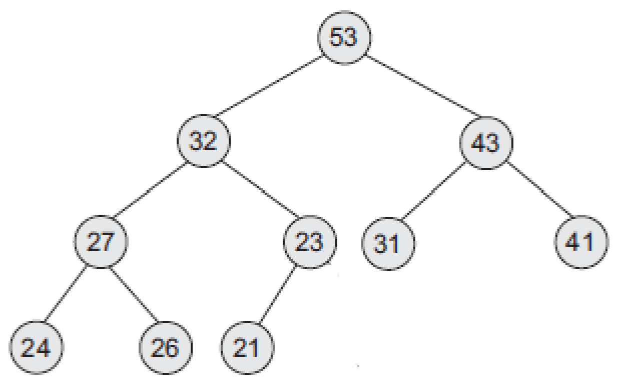
OPERATIONS ON HEAPS

The basic operations on heaps are listed as follows:

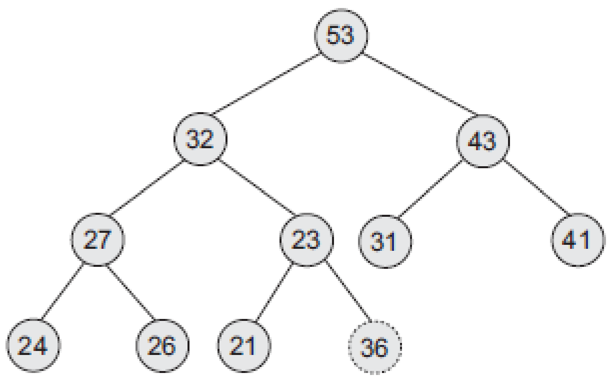
1. **Create** - creates an empty heap to which the root points
2. **Insert** - inserts an element into the heap
3. **Delete** - deletes max (or min) element from the heap
4. **ReheapUp** - rebuilds the heap when we use the insert() function
5. **ReheapDown** - rebuilds the heap when we use the delete() function

ReheapUp

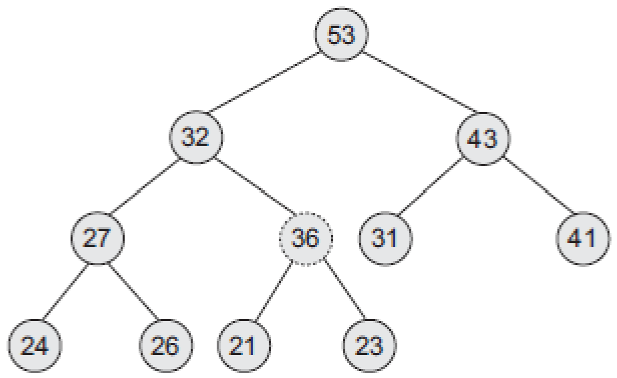
- Inserting a new node in the heap is always done at **the end of the heap** (leaf of last level).
- Sometimes the newly inserted node (the last node in the heap) becomes out of order.
- The **reheapUp** operation repairs the structure so that it is a heap by **lifting the last element up** in the tree until that element reaches a proper position in the tree.
- ReheapUp repairs a broken heap by lifting the last element up **by repeatedly exchanging child–parent keys** until it reaches the correct location in the heap.



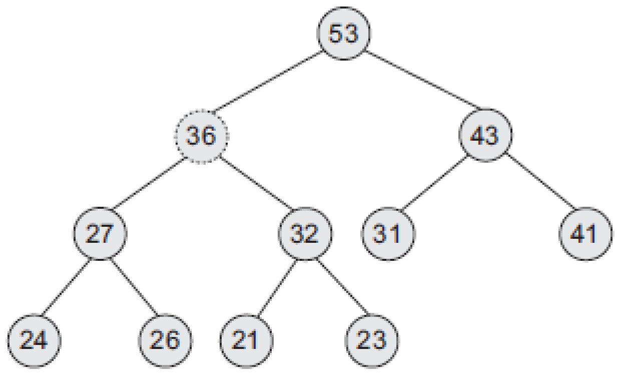
Original Heap



After Inserting 36



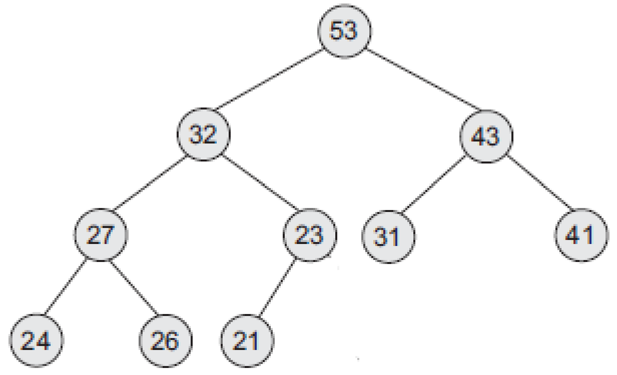
reheapUP
36 Exchanged with 23



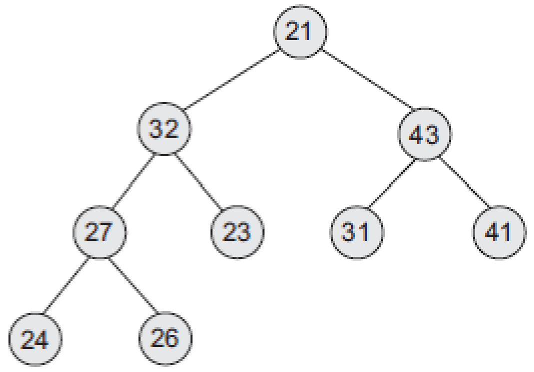
reheapUP
36 Exchanged with 32

ReheapDown

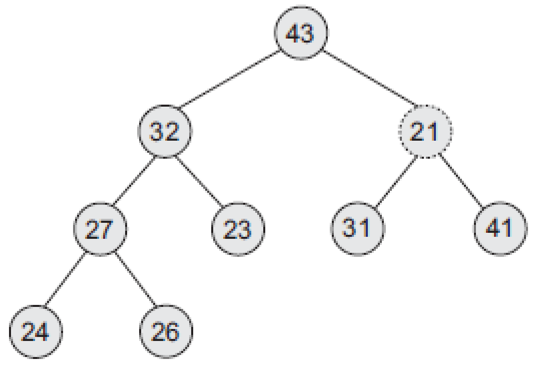
- Deleting a node in the heap is always done at **the top of the heap** (root). And then the last node in the heap replace the root position.
- Sometimes the new root becomes out of order.
- The **reheapDown** operation repairs the structure so that it is a heap by **lifting the root element down** in the tree until that element reaches a proper position in the tree.
- ReheapDown repairs a broken heap by lifting the last element down **by repeatedly exchanging parent-child keys** until it reaches the correct location in the heap.



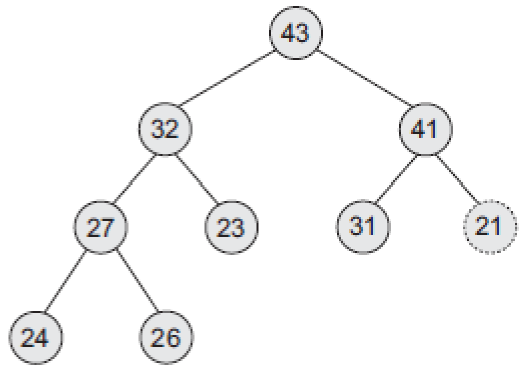
Original Heap



After deleting 53



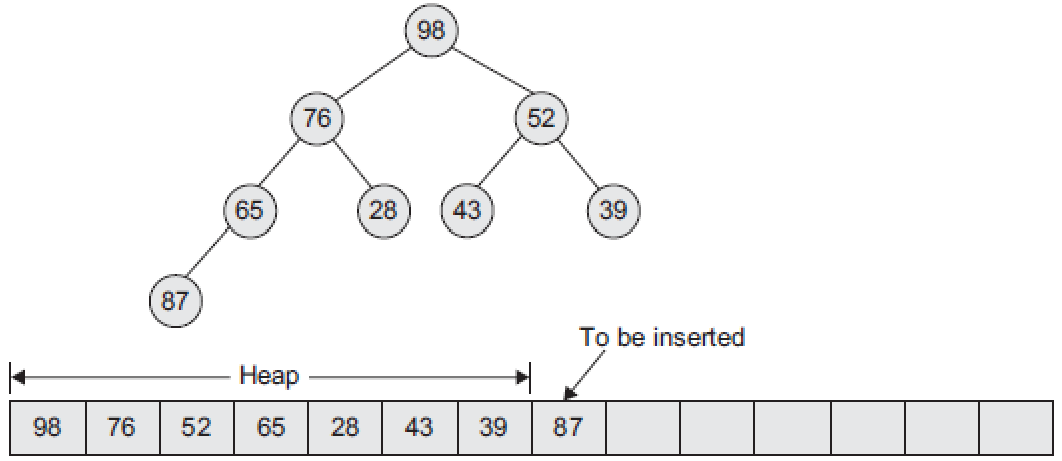
reheapDown
43 Exchanged with 21
Note: 43 > 32



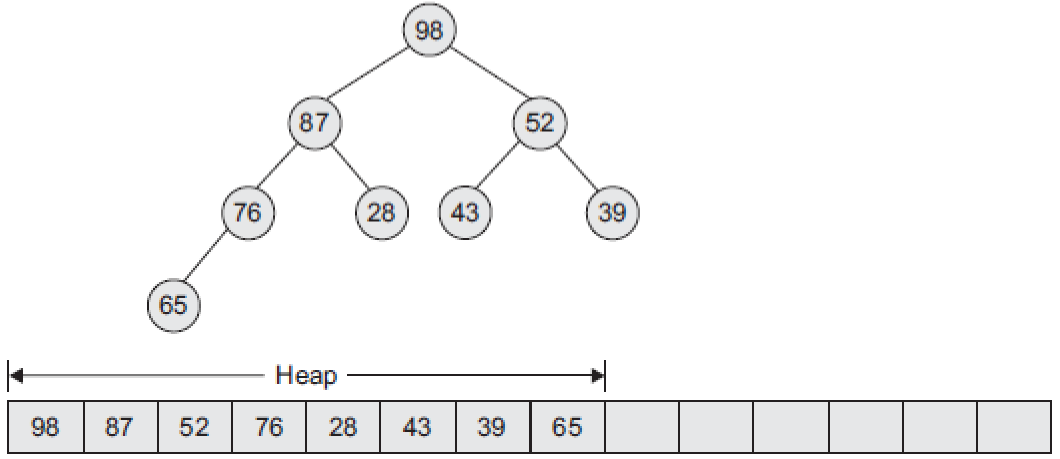
reheapDown
41 Exchanged with 21
Note: 41 > 31

INSERT

- A node can be inserted in a heap which has already been built, if there is an empty location in the array.
- To insert a node, we need to search the first empty leaf in the array. We find it immediately after the last node in the tree.
- To insert a node, we move the new data to the first empty leaf and perform reheapUp.



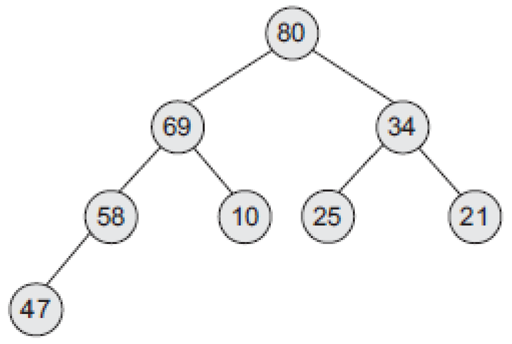
Heap after insertion of 87



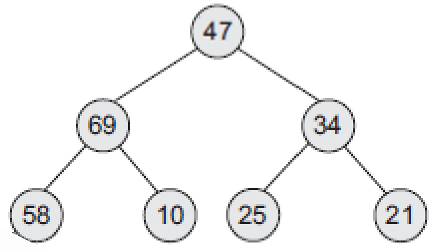
Heap after reheapUp Operation

DELETE

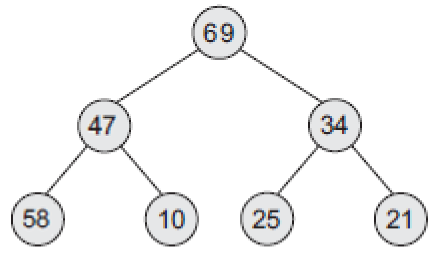
- While removing a node from a heap, the most common and meaningful logic is to delete the root.
  - The heap is thus left without a root.
- To reconstruct the heap, we move the data in the last heap node to the root and perform reheapDown.



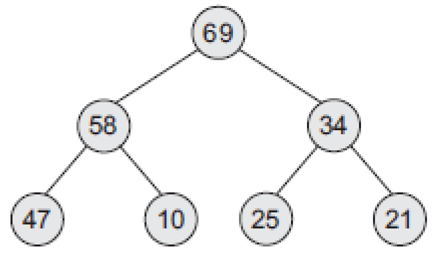
80	69	34	58	10	25	21	47	



47	69	34	58	10	25	21		



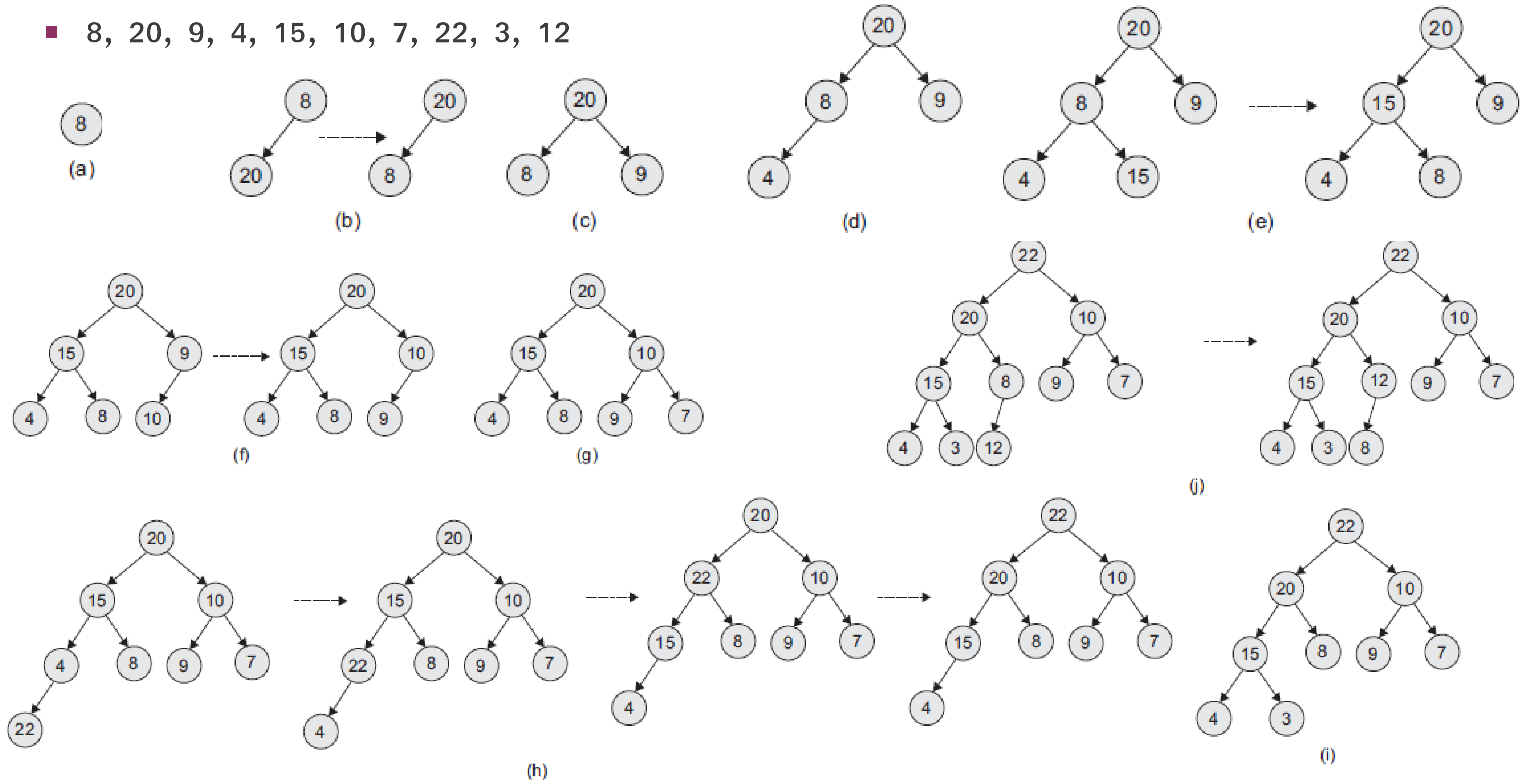
Heap after reheapDown Operation



Heap after reheapDown Operation

Create Heap for the following unsorted list of keys.

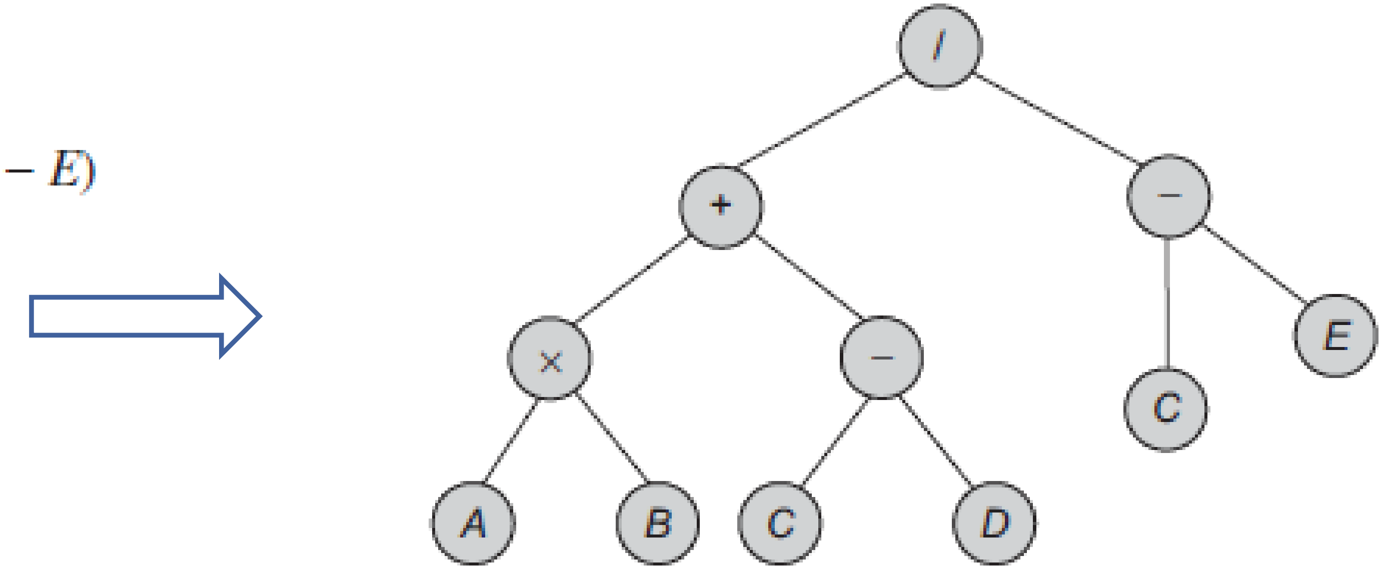
- 8, 20, 9, 4, 15, 10, 7, 22, 3, 12



EXPRESSION TREE

- A binary tree storing or representing an arithmetic expression is called as expression tree.
- The leaves of an expression tree are operands. Operands could be variables or constants.
- The branch nodes (internal nodes) represent the operators.
- A binary tree is the most suitable one for arithmetic expressions as it contains either binary or unary operators.

Let $E = ((A \times B) + (C - D))/(C - E)$



CONT..

- If we traverse this tree in **pre-order**, we visit the nodes in the order of: **/+xAB-CD-CE**, and this is a prefix form of the infix expression.
- On the other hand, if we traverse the tree in **post-order**, the nodes are visited in the following order: **ABxCD-E-/**, which is a postfix equivalent of the infix notation.

Let $E = ((A \times B) + (C - D))/(C - E)$