CSE-2321
Data Structure
Part-7

Presented by

Asmaul Hosna Sadika
Adjunct Faculty
Dept of CSE, IIUC

Contents

- Tree terminology
- Representation of binary trees in memory
- Traversing binary tree
- Binary search tree
- Insertion & deletion on binary search tree
- Heap
- Insertion & deletion on heap
- Heapsort
- B trees
- General tree; Balanced binary search tree (AVL tree, red black tree)

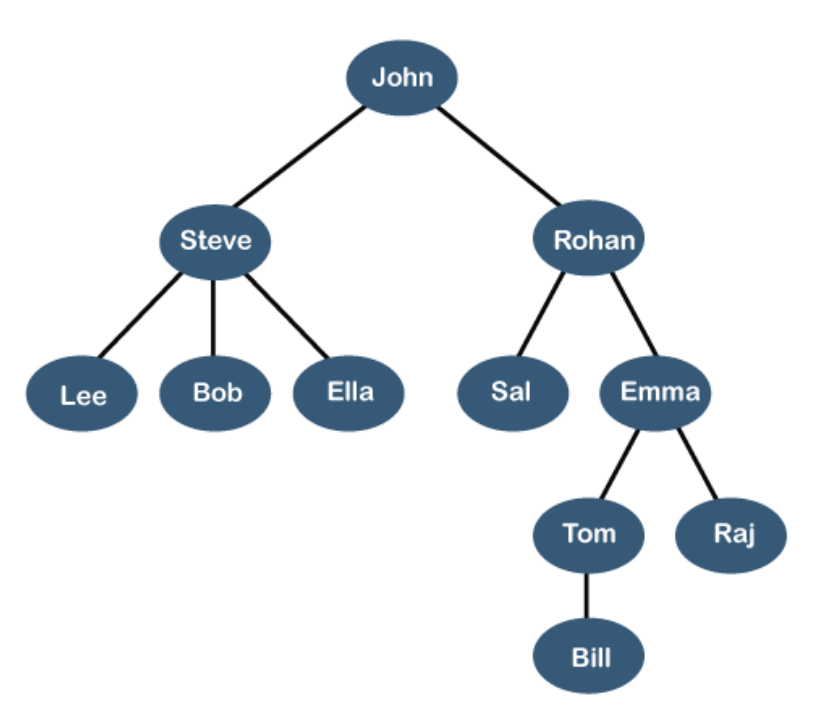
What is tree?

A tree is one of the non-linear data structures that represent hierarchical data.

Let's understand some key points of the Tree data structure.

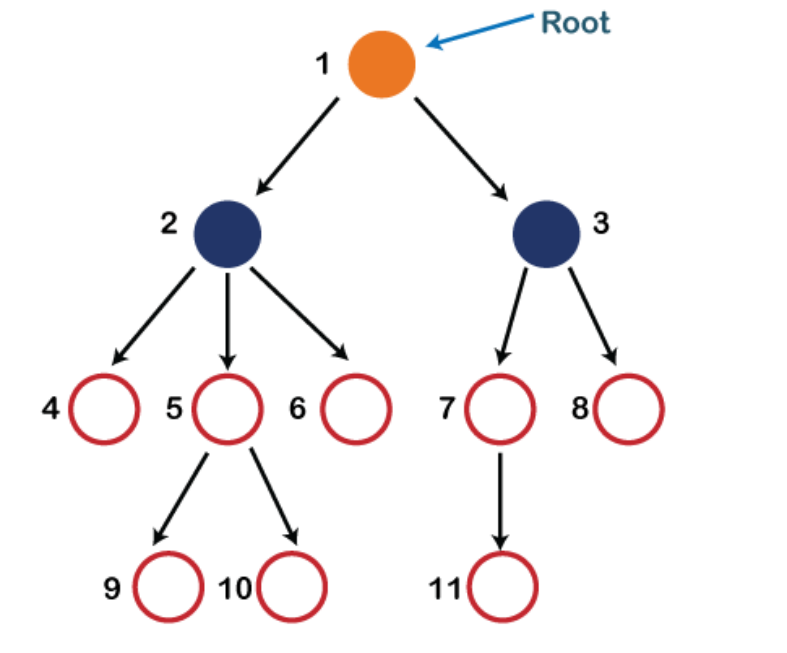
- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.
- In the Tree data structure, the topmost node is known as a root node. Each node contains some data, and data can be of any type. In the above tree structure, the node contains the name of the employee, so the type of data would be a string.
- Each node contains some data and the link or reference of other nodes that can be called children

Tree terminology



The above tree shows the **organization hierarchy** of some company. In the above structure, ***john*** is the **CEO** of the company, and John has two direct reports named as ***Steve*** and ***Rohan***. Steve has three direct reports named ***Lee, Bob, Ella*** where ***Steve*** is a manager. Bob has two direct reports named ***Sal*** and ***Emma***. **Emma** has two direct reports named ***Tom*** and ***Raj***. Tom has one direct report named ***Bill***. This particular logical structure is known as a ***Tree***. Its structure is similar to the real tree, so it is named a ***Tree***. In this structure, the ***root*** is at the top, and its branches are moving in a downward direction.

Some basic terms used in Tree data structure



In the structure, each node is labeled with some number. Each arrow shown in the above figure is known as a ***link*** between the two nodes.

Root: node numbered 1 is **the root node of the tree.**

Child node: If the node is a descendant of any node, then the node is known as a child node.

Parent: If the node contains any sub-node, then that node is said to be the parent of that sub-node.

Sibling: The nodes that have the same parent are known as siblings.

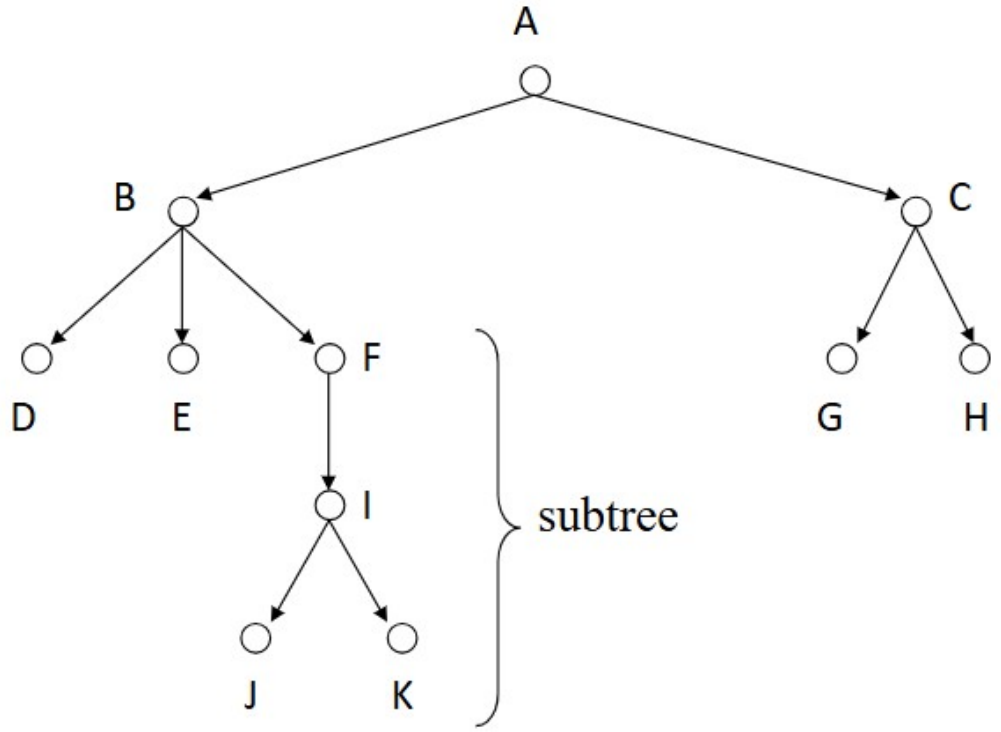
Leaf Node: The node of the tree, which doesn't have any child node, is called a leaf node. Leaf nodes can also be called external nodes.

Internal nodes: A node has at least one child node known as an ***internal***

Ancestor node: An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.

Descendant: The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

Example



- A is the ***root***
- D, E, G, H, J & K are ***leaves***
- B is the ***parent*** of D, E & F
- D, E & F are ***siblings*** and ***children*** of B
- I, J & K are ***descendants*** of B
- A & B are ***ancestors*** of I

Properties of Tree data structure

- **Recursive data structure:** The tree is also known as a ***recursive data structure***. A tree can be defined as recursively because the distinguished node in a tree data structure is known as a ***root node***.
- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have at least one incoming link known as an edge. There would be one link for the parent-child relationship.
- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.

Binary tree

The Binary tree means that the node can have maximum two children. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \ldots 2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

Binary tree

The minimum height can be computed as:

As we know that,

$n = 2^{h+1} - 1$

$n+1 = 2^{h+1}$

Taking log on both the sides,

$\log_2(n+1) = \log 2(2^{h+1})$

$\log_2(n+1) = h+1$

$h = \log_2(n+1) - 1$

The maximum height can be computed as:

As we know that,

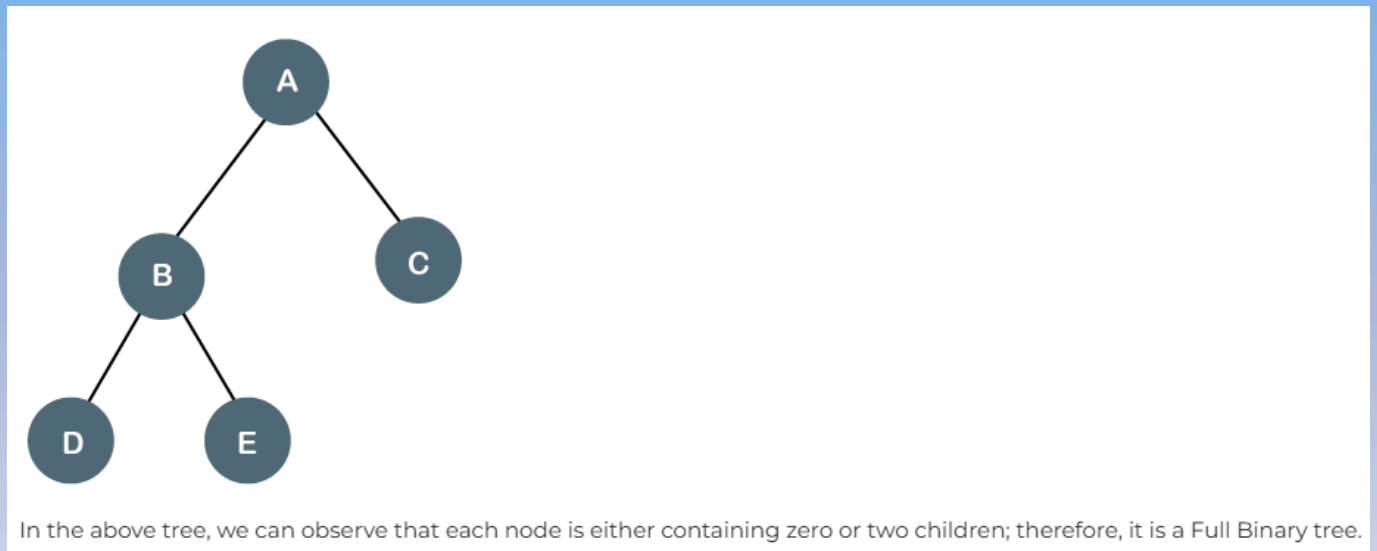
$n = h+1$

$h = n-1$

Types of Binary Tree

There are five types of Binary tree:

1. Full/ proper/ strict Binary tree
2. Complete Binary tree
3. Perfect Binary tree
4. Degenerate Binary tree
5. Balanced Binary tree



In the above tree, we can observe that each node is either containing zero or two children; therefore, it is a Full Binary tree.

Full/ proper/ strict Binary tree

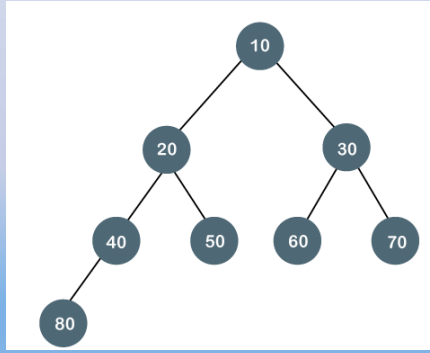
The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.

Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., 2^{h+1} -1.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of the full binary tree can be computed as: **h = n+1/2**

Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

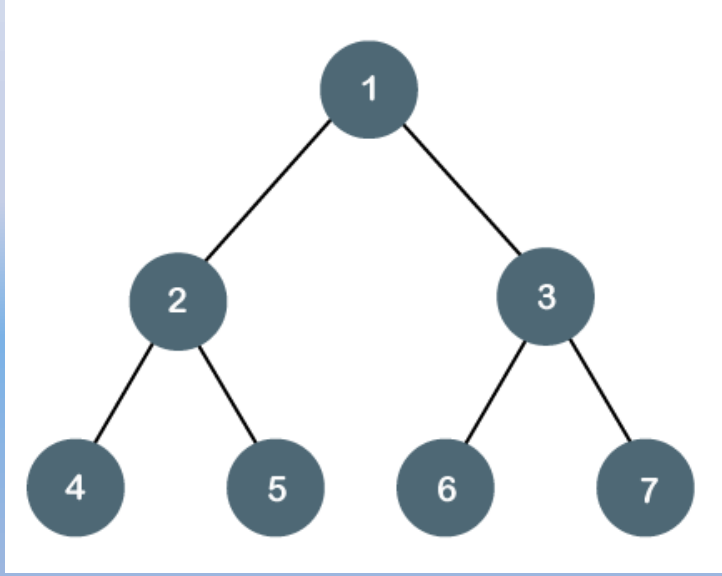


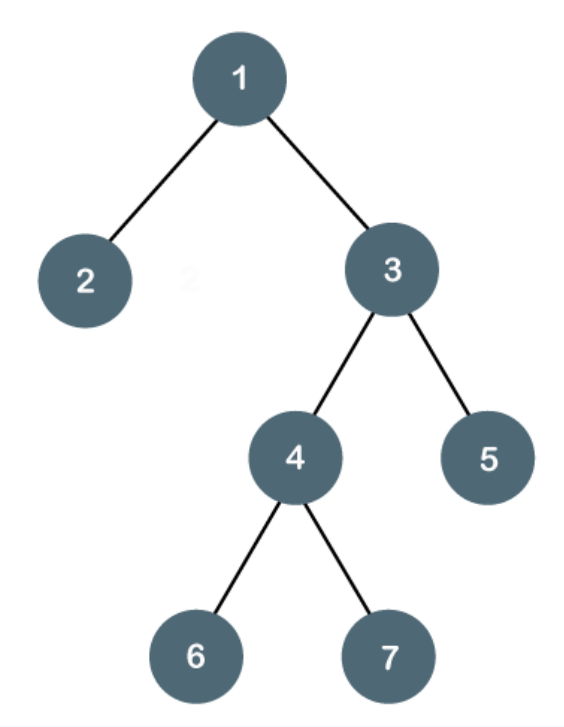
Properties of Complete Binary Tree

- The maximum number of nodes in complete binary tree is $2^{h+1} - 1$.
- The minimum number of nodes in complete binary tree is 2^{h}.
- The minimum height of a complete binary tree is **$\log_2(n+1) - 1$.**
- The maximum height of a complete binary tree is

Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.

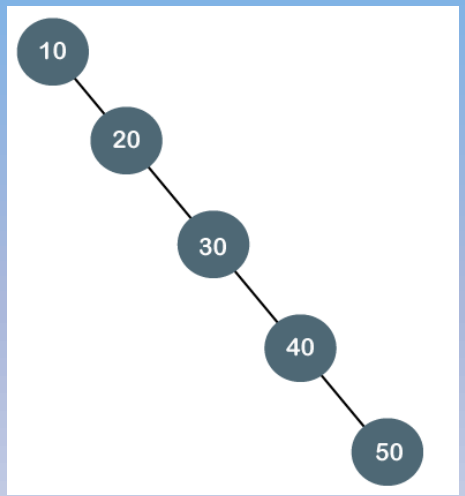


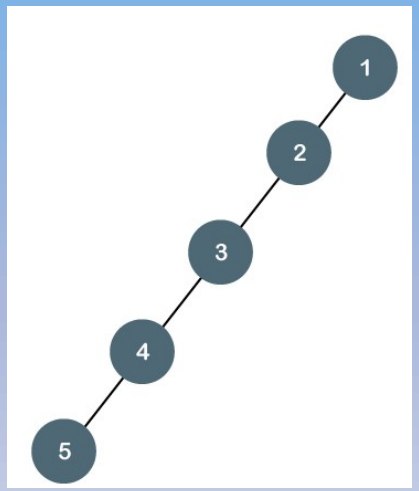


Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one child.

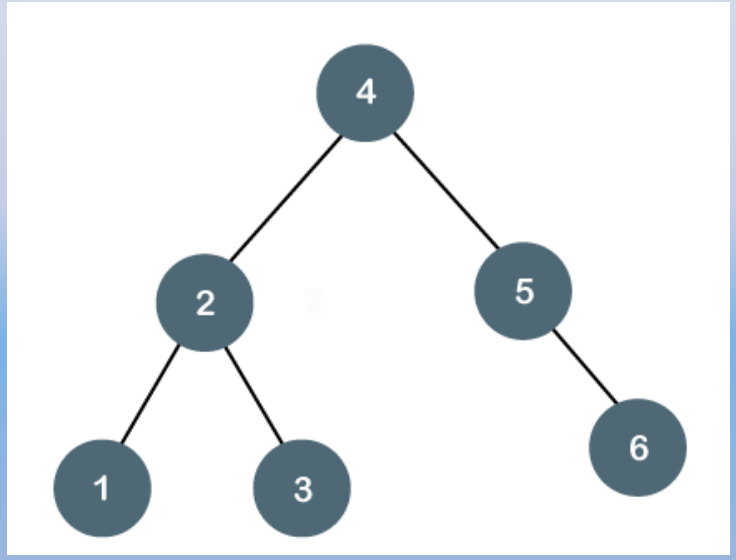
- The left side tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.
- The right side tree is also a degenerate binary tree because all the nodes have only one child. It is also known as a left-skewed tree as all the nodes have a left child only.





Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by at most 1. For example, ***AVL*** and ***Red-Black trees*** are balanced binary tree.



The above tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.

Representation of binary trees in memory

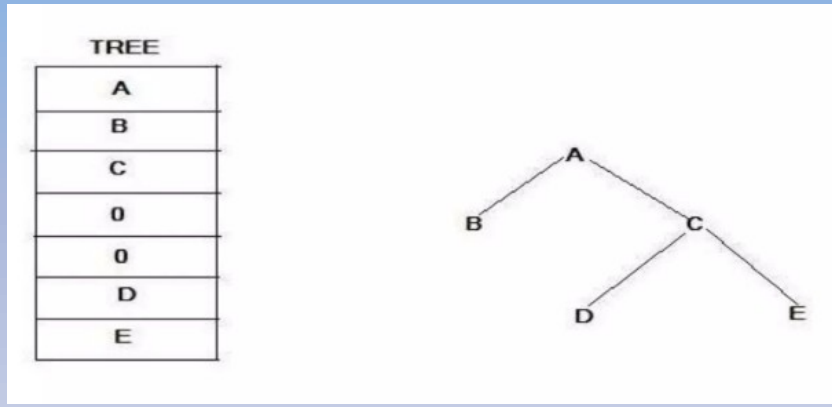
There are two ways to represent the tree in memory:

1. Sequential representation
2. Linked representation

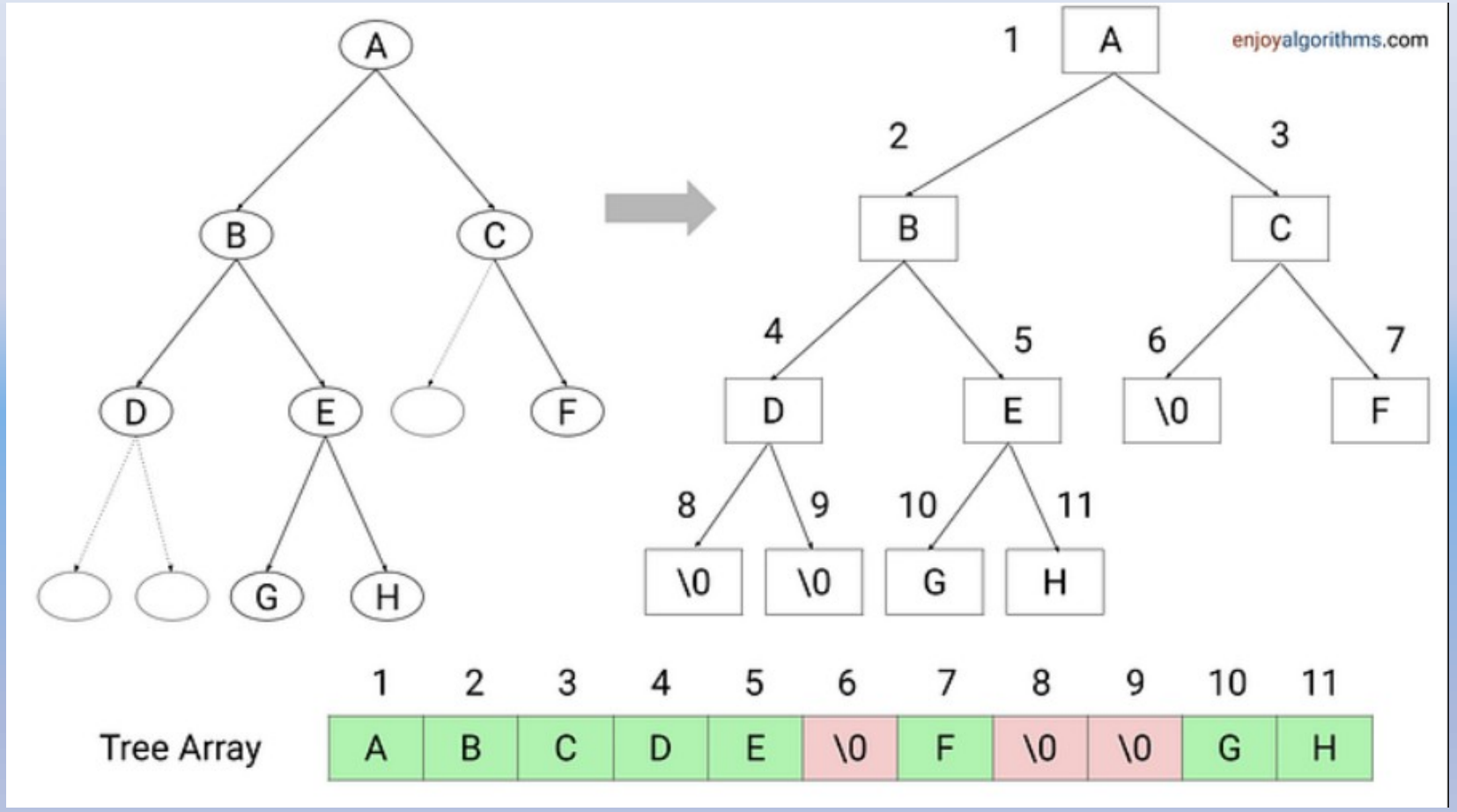
Sequential Representation

Suppose T is a complete binary tree then only single linear array TREE is used as follows

- The root is stored in TREE[1]
- If a node n occupies TREE[K], then its left child in TREE[2*K] and right child in TREE[2*K+1]
- If TREE[1]= NULL then it is empty tree



Sequential Representation

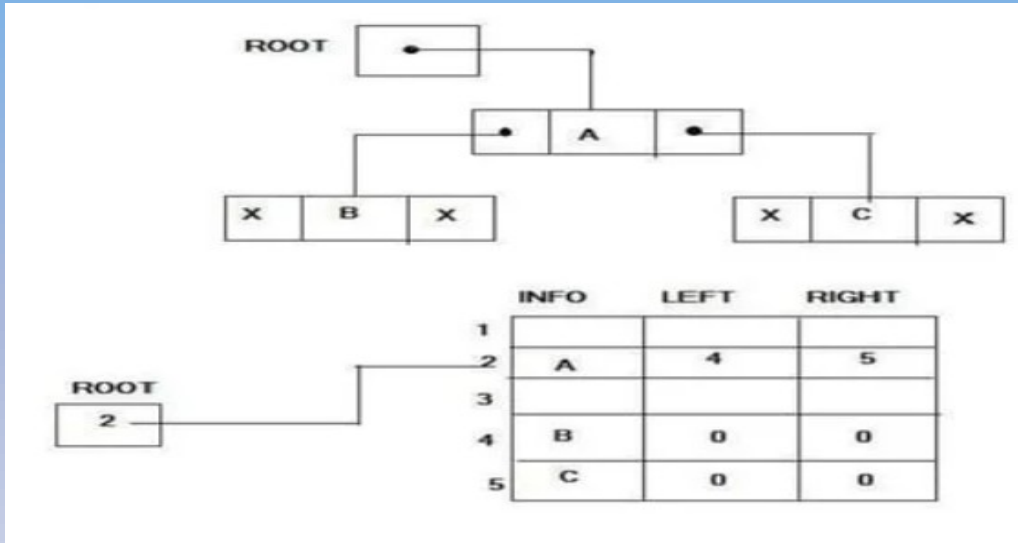


Linked representation

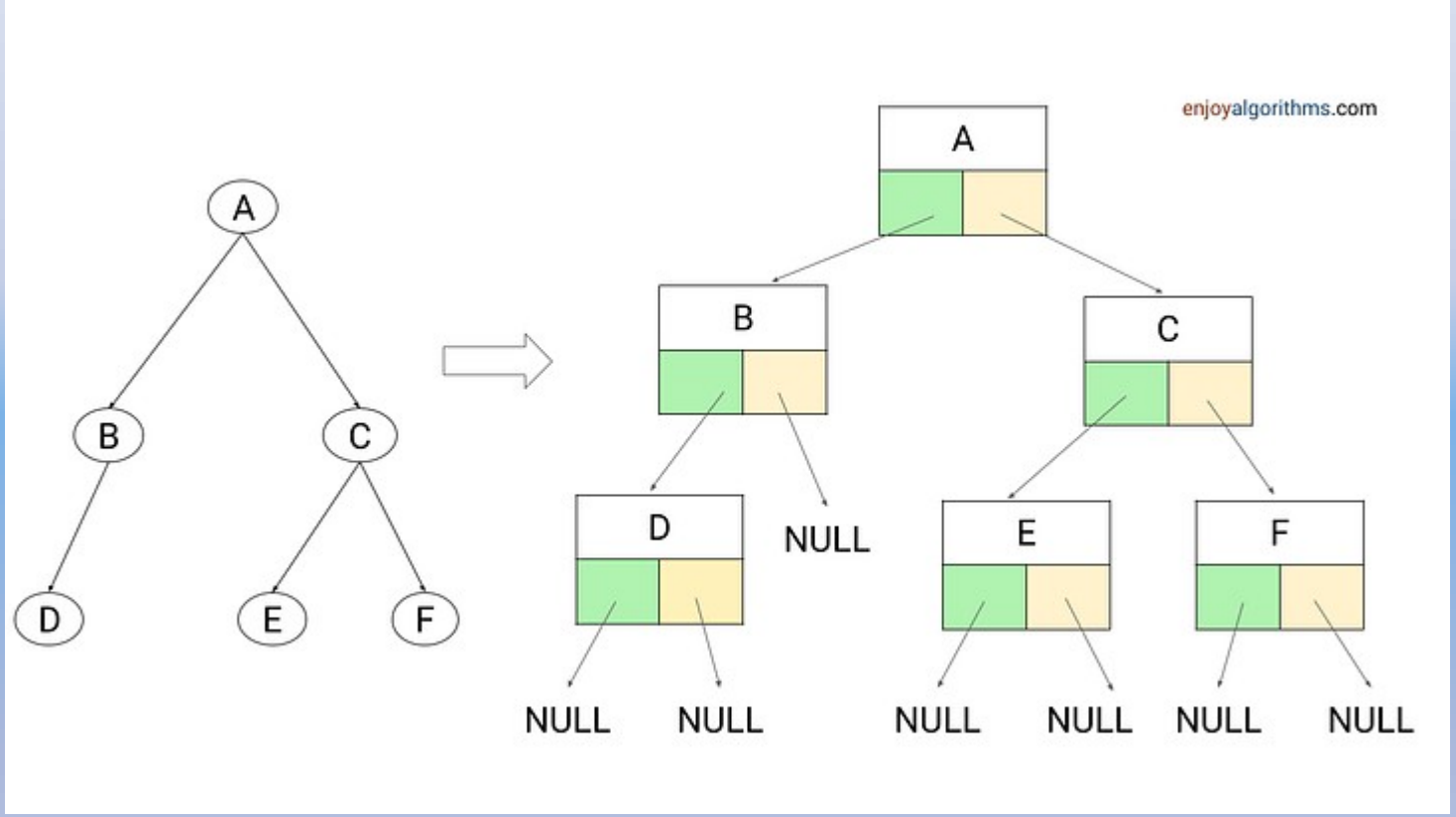
The linked representation uses three parallel arrays INFO, LEFT & RIGHT and a pointer variable ROOT. Each node N and Y will correspond to a location K such that:

1. INFO[K] contains data at node N
2. LEFT[K] contains the location of left child of node N
3. RIGHT[K] contains the location of right child of node N

ROOT will contain location of R and T



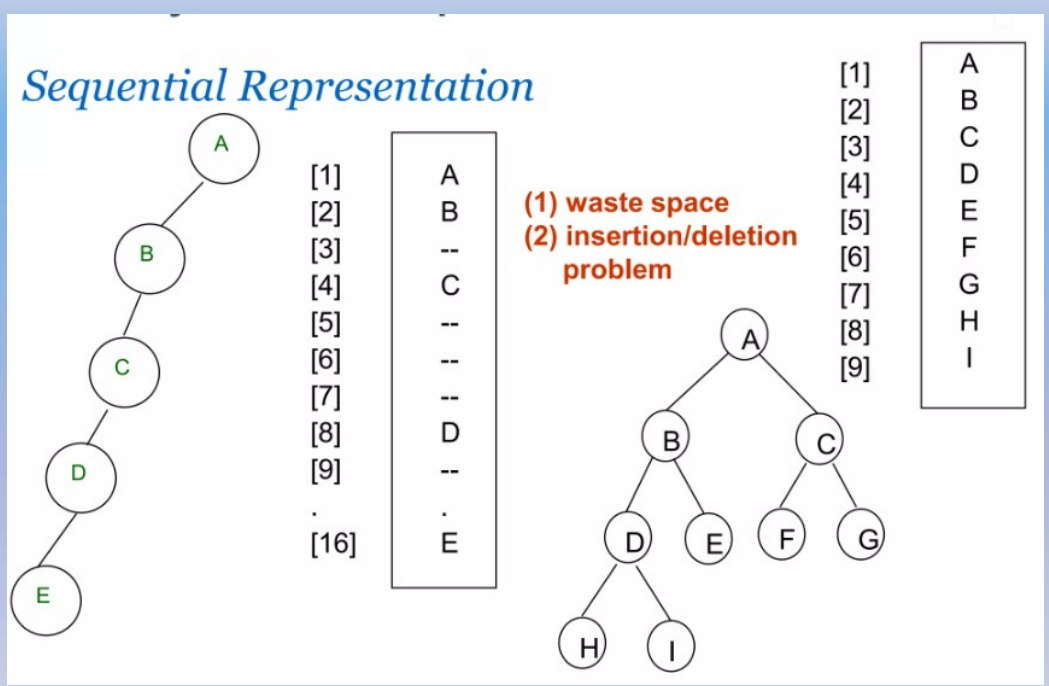
Linked representation



Representation of binary trees in memory

Array representation is good for complete binary tree, but it is wasteful for many other binary trees.

The representation suffers from insertion and deletion of node from the middle of the tree, as it requires the moment of potentially many nodes to reflect the change in level number of this node.



To overcome this difficulty we represent the binary tree in linked representation.

The advantage of using linked representation of binary tree is that:

- Insertion and deletion involve no data movement and no movement of nodes except the rearrangement of pointers.

The disadvantages of linked representation of binary tree includes:

- Given a node structure, it is difficult to determine its parent node.
- Memory spaces are wasted for storing NULL pointers for the nodes, which have no subtrees.

Tree traversal

Traversal is a process to visit all the nodes of a tree and may print their values too.

Because all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree.

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.

Binary Tree Traversal

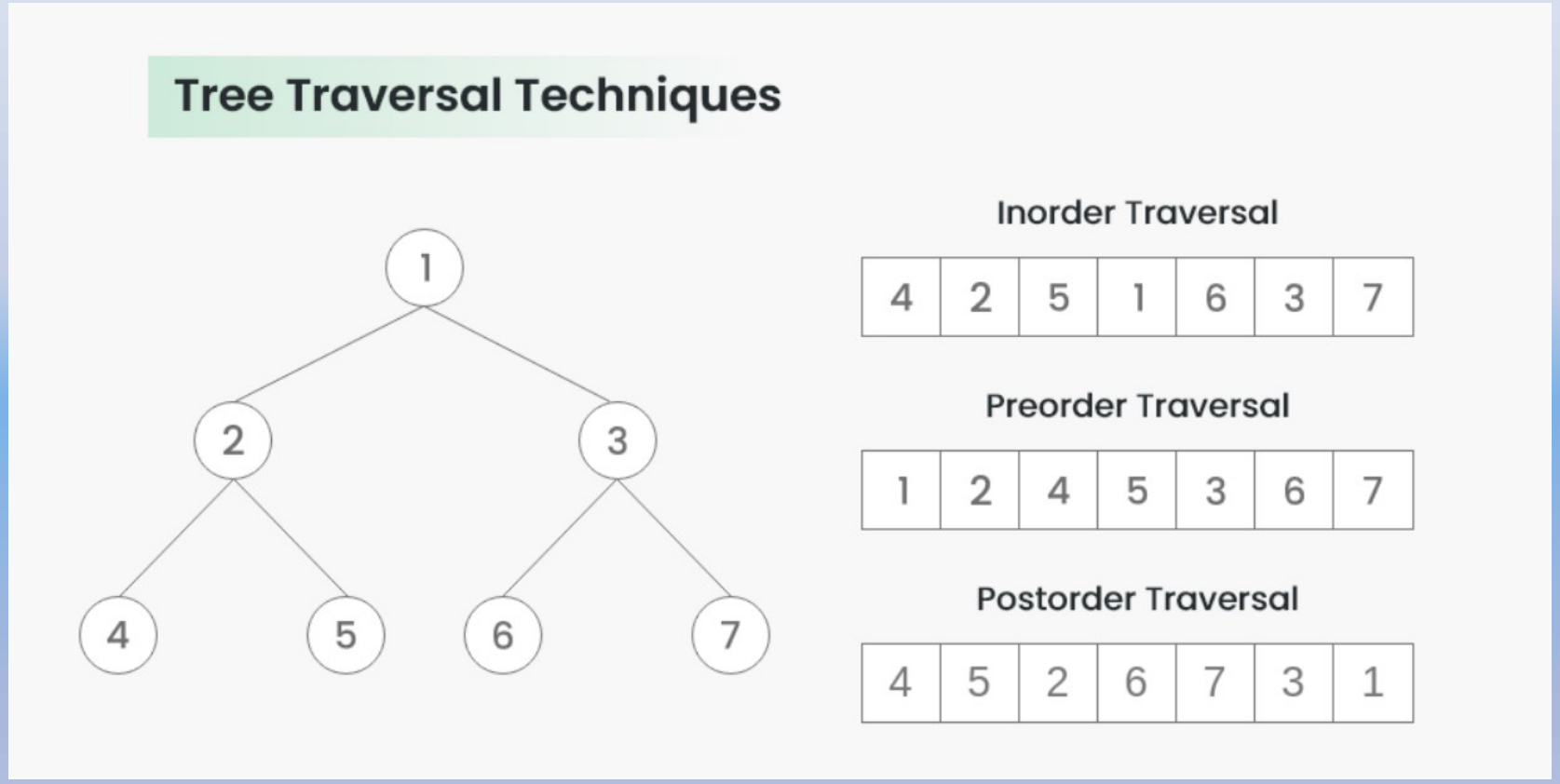
A traversal of a tree T is a systematic way of accessing or visiting all the node of T.

There are three standard ways of traversing a binary tree T with root R.

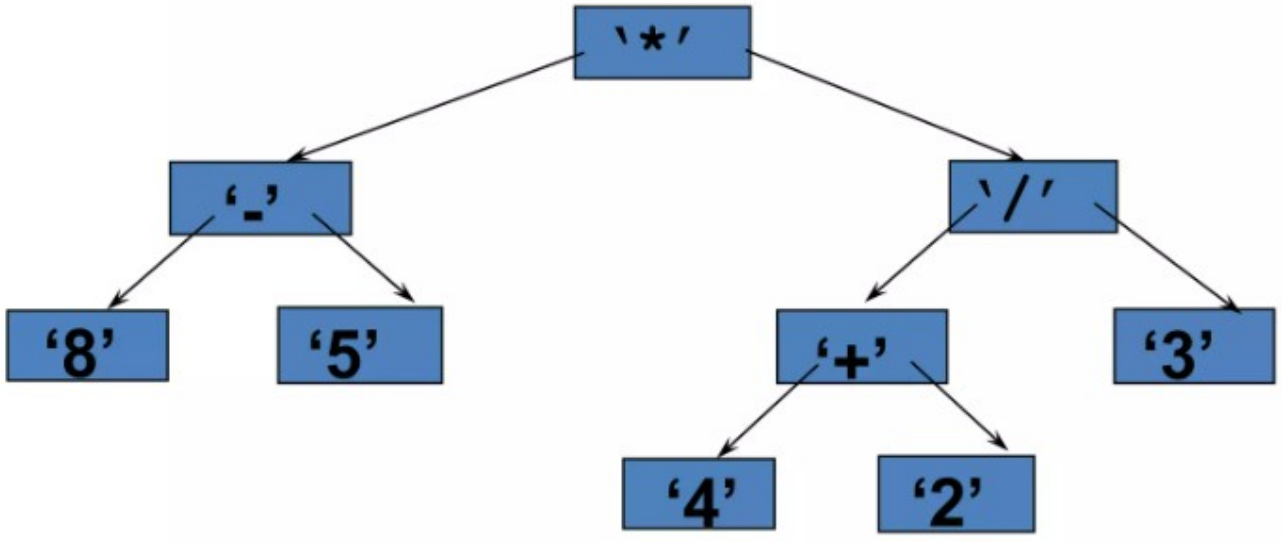
These are :

1. **Preorder (N L R):**
    a) Process the node/root.
    b) Traverse the Left sub tree.
    c) Traverse the Right sub tree.
2. **Inorder (L N R):**
    a) Traverse the Left sub tree.
    b) Process the node/root.
    c) Traverse the Right sub tree.
3. **Postorder (L R N):**
    a) Traverse the Left sub tree.
    b) Traverse the Right sub tree.
    c) Process the node/root

Binary Tree Traversal



Preparing a Tree from an infix arithmetic expression

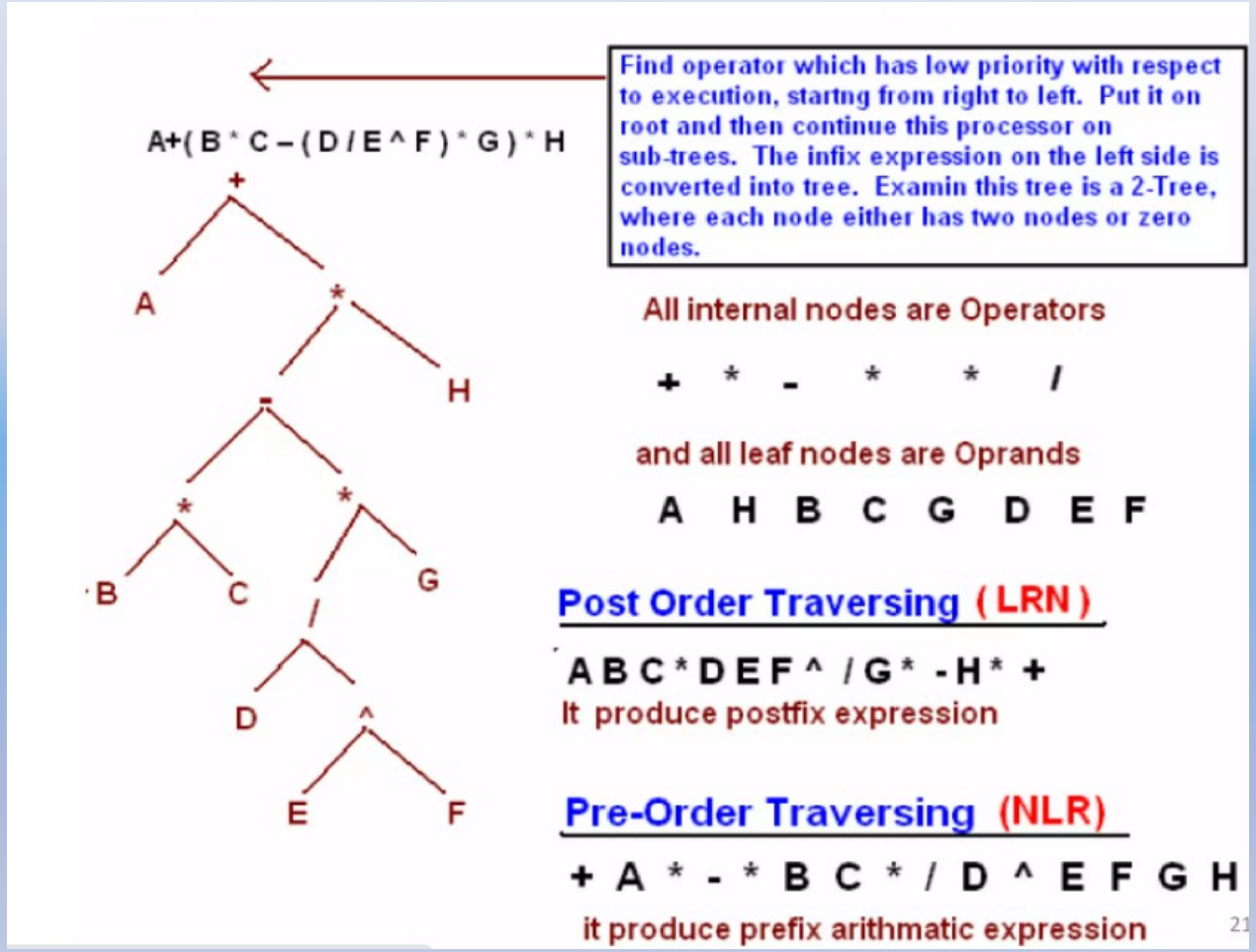


Infix: ((8 - 5) * ((4 + 2) / 3))

Prefix: * - 8 5 / + 4 2 3

Postfix: 8 5 - 4 2 + 3 / *

Preparing a Tree from an infix arithmetic expression



Formation of Binary Tree from its Traversal

7.2 A binary tree T has 9 nodes. The inorder and preorder traversals of T yield the followi sequences of nodes:

Inorder:	E	A	C	K	F	H	D	B	G
Preorder:	F	A	E	K	C	D	H	G	B

Draw the tree T.

The tree T is drawn from its root downward as follows.

(a) The root of T is obtained by choosing the first node in its preorder. Thus F is the root of T.

(b) The left child of the node F is obtained as follows. First use the inorder of T to find the nodes in the left subtree T_1 of F. Thus T_1 consists of the nodes E, A, C and K. Then the left child of F is obtained by choosing the first node in the preorder of T_1 (which appears in the preorder of T). Thus A is the left son of F.

(c) Similarly, the right subtree T_2 of F consists of the nodes H, D, B and G, and D is the root of T_2, that is, D is the right child of F.

Repeating the above process with each new node, we finally obtain the required tree in Fig. 7.74.

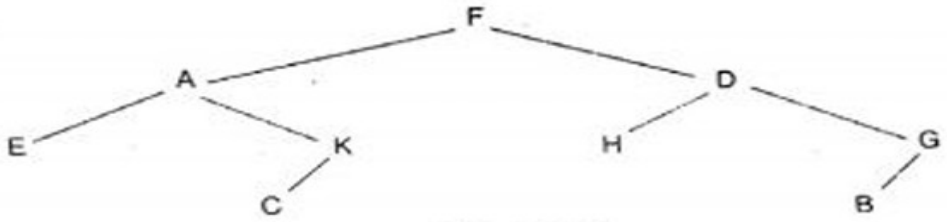
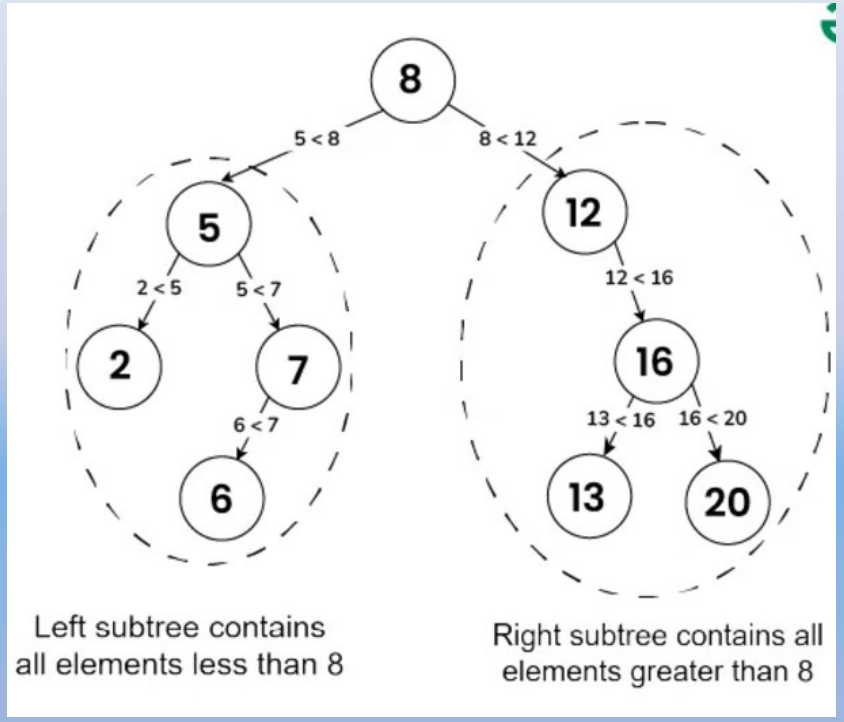


Fig. 7.74

Binary Search Tree

- **Binary Search Tree** is a data structure used in computer science for organizing and storing data in a sorted manner.
- Binary search tree follows all properties of binary tree and for every nodes, its **left** subtree contains values less than the node and the **right** subtree contains values greater than the node.
- This hierarchical structure allows for efficient **Searching**, **Insertion**, and **Deletion** operations on the data stored in the tree.



Binary Search Tree

Properties of Binary Search Tree:

- The left subtree of a node contains only nodes with keys lesser than the node’s key.
- The right subtree of a node contains only nodes with keys greater than the node’s key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes(BST may have duplicate values with different handling approaches).

Suppose an ITEM of information is given. The following algorithm finds the location of ITEM in the binary search tree T, or inserts ITEM as a new node in its appropriate place in the tree.

(a) Compare ITEM with the root node N of the tree.

- **(i)** If ITEM < N, proceed to the left child of N.
- **(ii)** If ITEM > N, proceed to the right child of N.

(b) Repeat Step (a) until one of the following occurs:

- **(i)** We meet a node N such that ITEM = N. In this case the search is successful.
- **(ii)** We meet an empty subtree, which indicates that the search is unsuccessful, and we insert ITEM in place of the empty subtree.

In other words, proceed from the root R down through the tree T until finding ITEM in T or inserting ITEM as a terminal node in T.

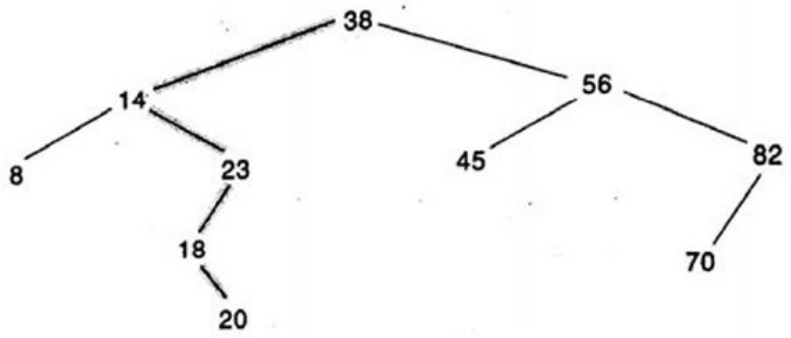
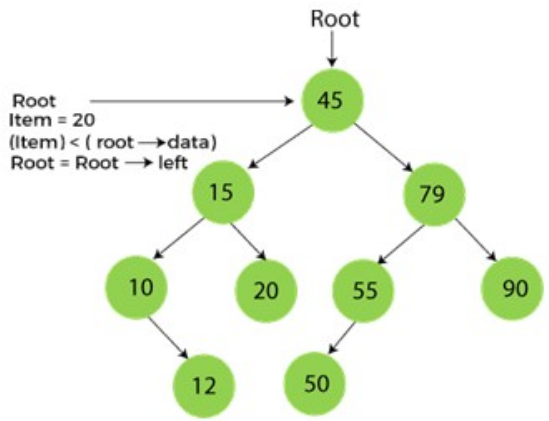


Fig. 7.22 *ITEM = 20 Inserted*

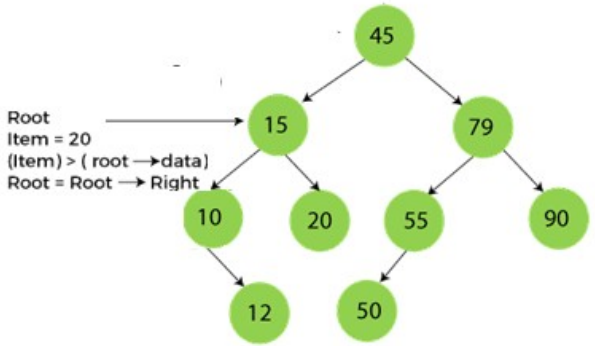
Searching for some specific node in binary search tree is pretty easy due to the fact that, element in BST are stored in a particular order.

- Compare the element with the root of the tree.
- If the item is matched then return the location of the node.
- Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
- If not, then move to the right sub-tree.
- Repeat this procedure recursively until match found.
- If element is not found then return NULL.

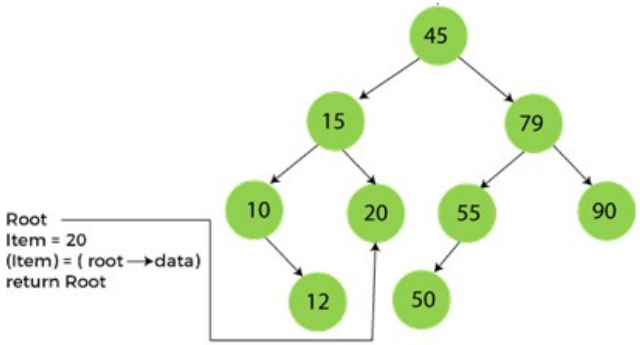
Step1:



Step2:



Step3:



Searching

FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)

A binary search tree T is in memory, and an $ITEM$ of information is given.
This procedure finds the location LOC of $ITEM$ in T, and also the location PAR of the parent of $ITEM$.

There are three special cases:

1. $LOC = NULL$ and $PAR = NULL$ indicates the tree is empty.
2. $LOC \neq NULL$ and $PAR = NULL$ indicates $ITEM$ is the root of T.
3. $LOC = NULL$ and $PAR \neq NULL$ indicates $ITEM$ is not in T, and can be added to T as a child of the node N at location PAR.

Algorithm Steps:

1. **[Tree empty?]**
   If $ROOT = NULL$, then:
   $LOC := NULL$, $PAR := NULL$, and Return.
2. **[ITEM at root?]**
   If $ITEM = INFO[ROOT]$, then:
   $LOC := ROOT$, $PAR := NULL$, and Return.
3. **[Initialize pointers PTR and $SAVE$]**
   If $ITEM < INFO[ROOT]$, then:
   $PTR := LEFT[ROOT]$, $SAVE := ROOT$.
   Else:
   $PTR := RIGHT[ROOT]$, $SAVE := ROOT$.
   [End of IF structure]
4. Repeat Steps 5 and 6 while $PTR \neq NULL$:
5. **[ITEM found?]**
   If $ITEM = INFO[PTR]$, then:
   $LOC := PTR$, $PAR := SAVE$, and Return.
6. If $ITEM < INFO[PTR]$, then:
   $SAVE := PTR$, $PTR := LEFT[PTR]$.
   Else:
   $SAVE := PTR$, $PTR := RIGHT[PTR]$.
   [End of IF structure]
   [End of Step 4 loop]
7. **[Search unsuccessful]**
   Set $LOC := NULL$, $PAR := SAVE$.
8. **Exit**

Insertion

INSBST(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)

A binary search tree T is in memory, and an $ITEM$ of information is given.

This algorithm finds the location LOC of $ITEM$ in T or adds $ITEM$ as a new node in T at location LOC.

Algorithm Steps:

1. **Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)**
   (Refer to Procedure 7.4).
2. **If $LOC \neq NULL$, then Exit.**
3. **[Copy $ITEM$ into a new node in the $AVAIL$ list.]**
   (a) If $AVAIL = NULL$, then:
   Write: OVERFLOW, and Exit.
   (b) Set $NEW := AVAIL$, $AVAIL := LEFT[AVAIL]$, and $INFO[NEW] := ITEM$.
   (c) Set $LOC := NEW$, $LEFT[NEW] := NULL$, and $RIGHT[NEW] := NULL$.
4. **[Add $ITEM$ to the tree.]**
   If $PAR = NULL$, then:
   $ROOT := NEW$.
   Else if $ITEM < INFO[PAR]$, then:
   $LEFT[PAR] := NEW$.
   Else:
   $RIGHT[PAR] := NEW$.
   [End of IF structure]
5. **Exit.**

What is Heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have utmost two children. Before knowing more about the heap data structure, we should know about the complete binary tree.

What is a complete binary tree?

A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

The heap tree is a special balanced binary tree data structure where the root node is compared with its children and arrange accordingly.

How can we arrange the nodes in the Tree?

There are two types of the heap:

- Min Heap
- Max heap

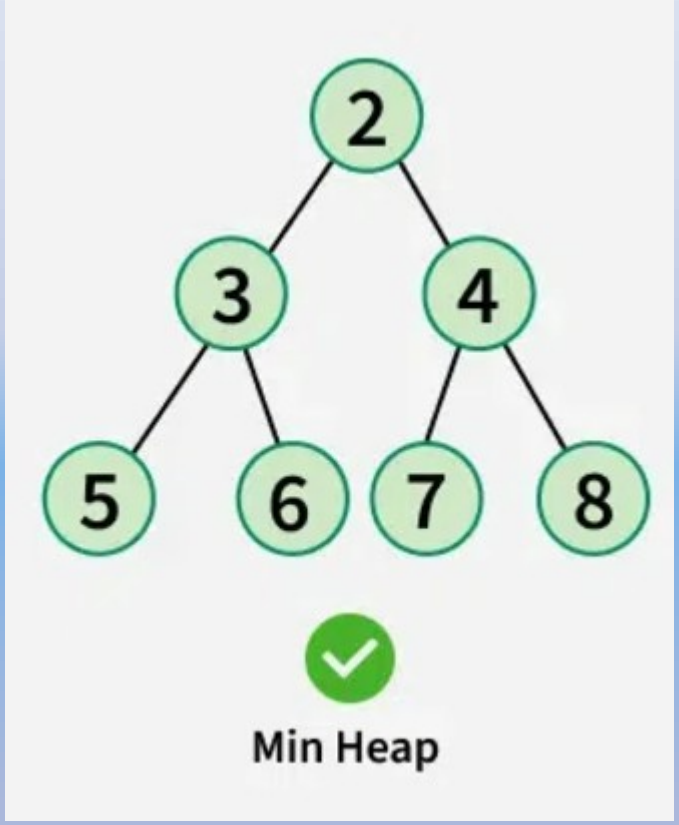
Min Heap: The value of the parent node should be less than or equal to either of its children.

Or

In other words, the min-heap can be defined as, for every node i, the value of node i is greater than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] <= A[i]

Min heap



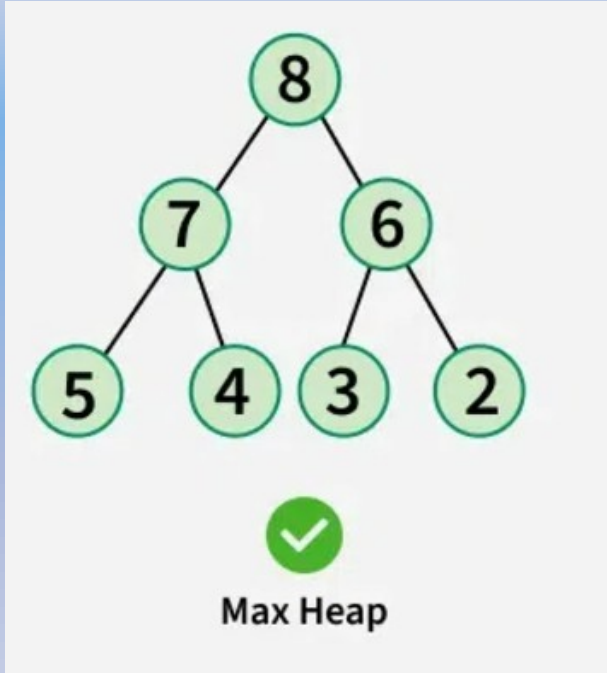
Max Heap

Max Heap: The value of the parent node is greater than or equal to its children.

Or

In other words, the max heap can be defined as for every node i; the value of node i is less than or equal to its parent value except the root node. Mathematically, it can be defined as:

A[Parent(i)] >= A[i]



Array representation of Binary Heap

A Binary Heap is a Complete Binary Tree. A binary heap is typically represented as array. The representation is done as:

- The root element will be at Arr[0].
- Below table shows indexes of other nodes for the ith node, i.e., Arr[i]

Arr[(i-1)/2]	Returns the parent node
Arr[(2*i)+1]	Returns the left child node
Arr[(2*i)+2]	Returns the right child node

Heap Insertion

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.

[**Note** − In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.]

Suppose we want to build a heap H from the following list of numbers:

44, 30, 50, 22, 60, 55, 77, 55

This can be accomplished by inserting the eight numbers one after the other into an empty heap H using the above procedure. Figure 7.60(a) through (h) shows the respective pictures of the heap after each of the eight elements has been inserted. Again, the dotted line indicates that an exchange has taken place during the insertion of the given ITEM of information.

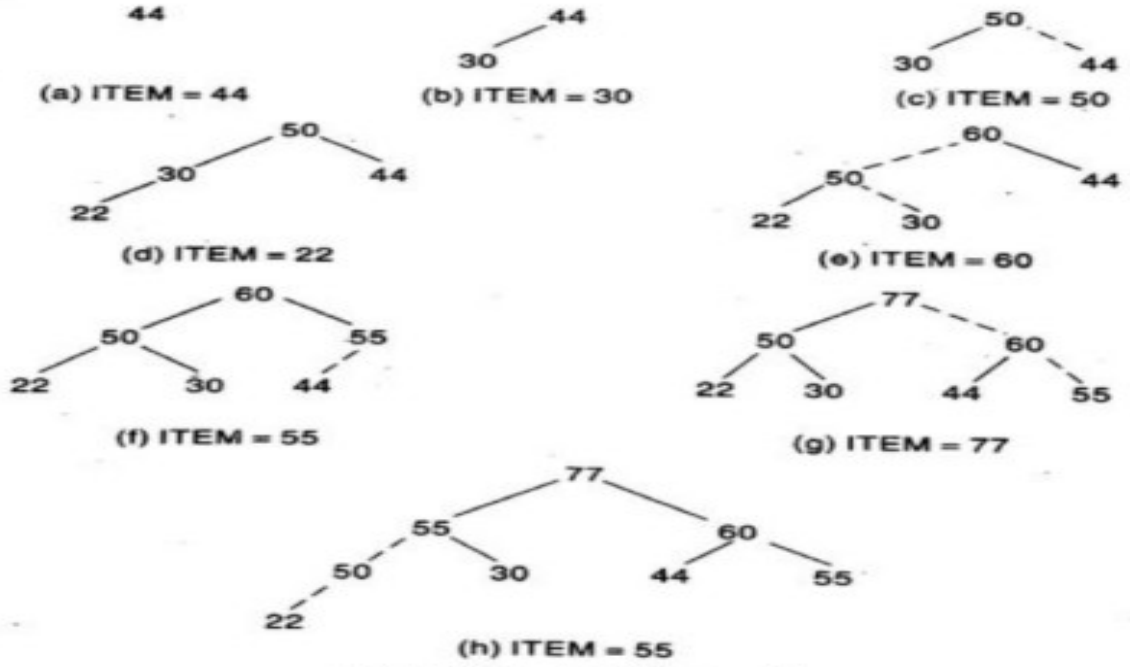


Fig. 7.60 *Building a Heap*

Procedure 7.9: INSHEAP(TREE, N, ITEM)

A heap H with N elements is stored in the array TREE, and an ITEM of information is given. This procedure inserts ITEM as a new element of H. PTR gives the location of ITEM as it rises in the tree, and PAR denotes the location of the parent of ITEM.

1. [Add new node to H and initialize PTR.]
   Set N := N + 1 and PTR := N.
2. [Find location to insert ITEM.]
   Repeat Steps 3 to 6 while PTR < 1.
3. Set PAR := ⌊PTR/2⌋. [Location of parent node.]
4. If ITEM ≤ TREE[PAR], then:
   Set TREE[PTR] := ITEM, and Return.
   [End of If structure.]
5. Set TREE[PTR] := TREE[PAR]. [Moves node down.]
6. Set PTR := PAR. [Updates PTR.]
   [End of Step 2 loop.]
7. [Assign ITEM as the root of H.]
   Set TREE[1] := ITEM.
8. Return.

Deletion in Heap Tree

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.

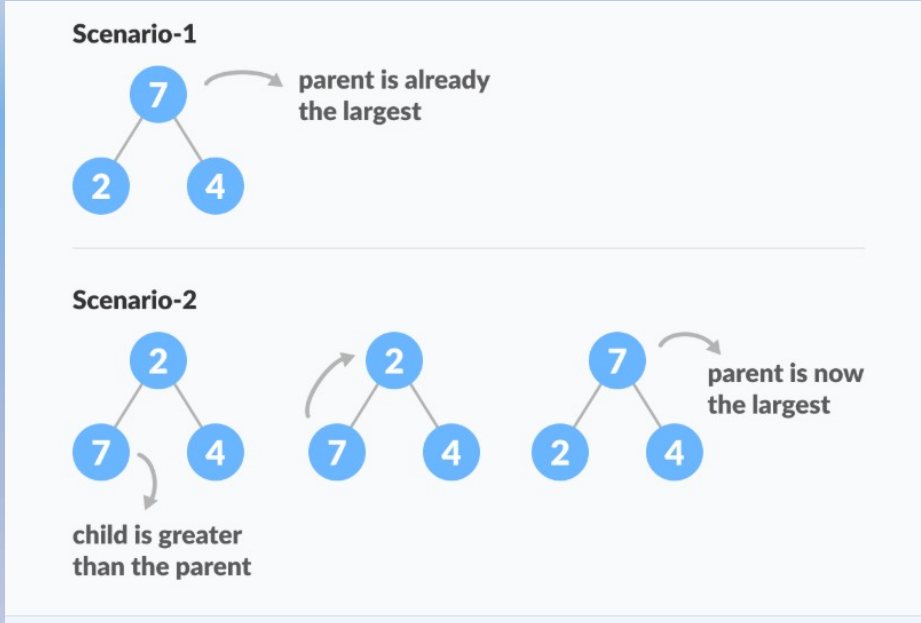
Step 4 − If value of parent is less than child, then swap them.

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

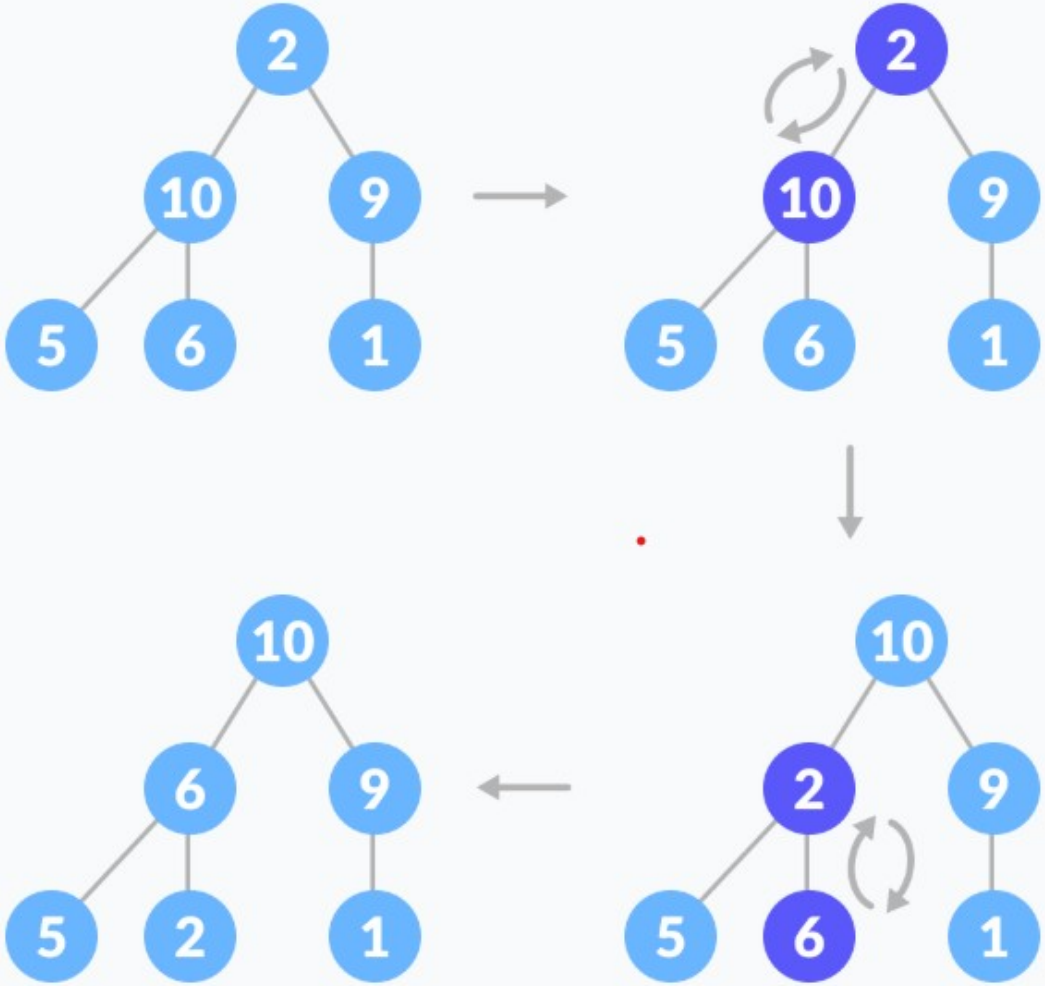
How to "heapify" a tree

Starting from a complete binary tree, we can modify it to become a Max-Heap by running a function called heapify on all the non-leaf elements of the heap.

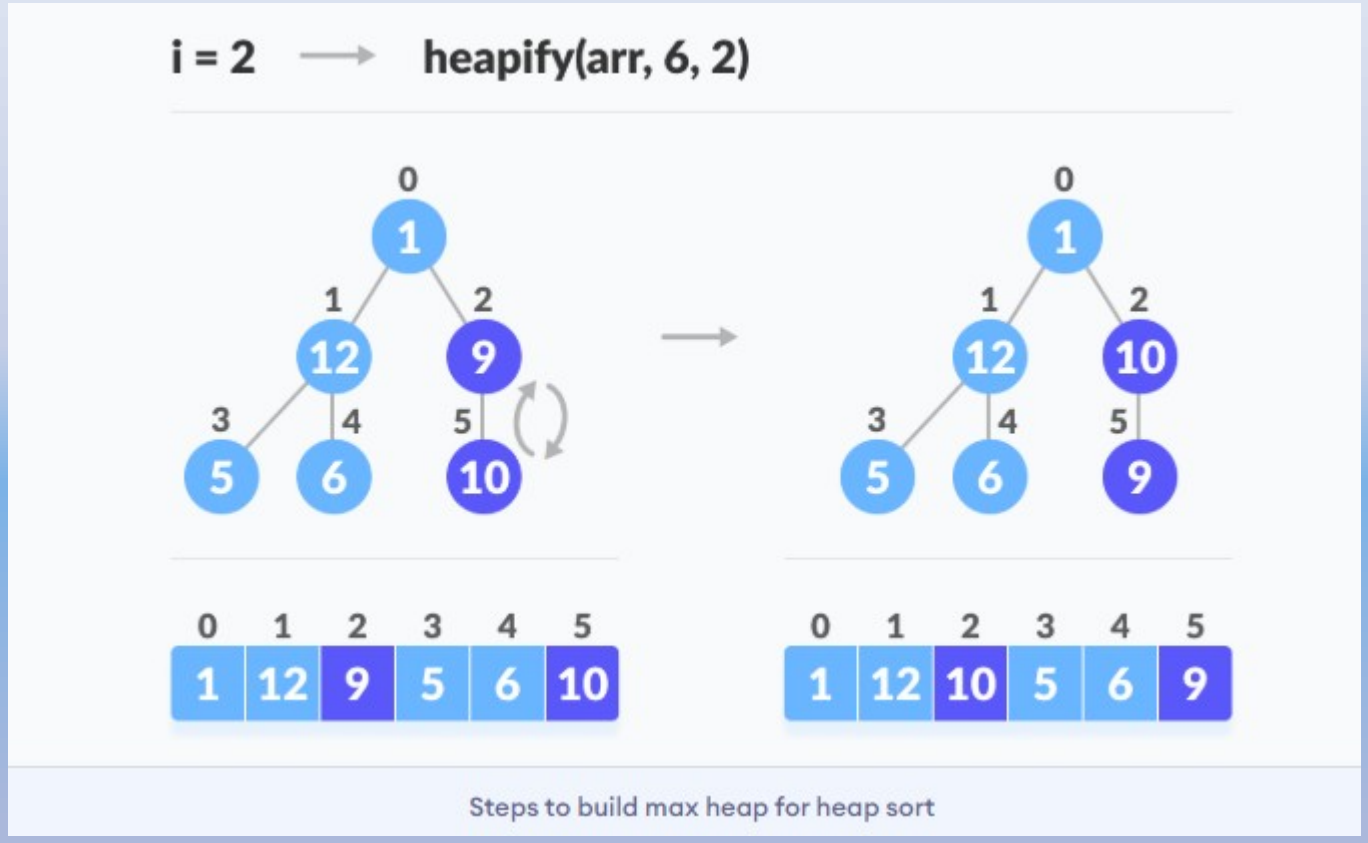
Since heapify uses recursion, it can be difficult to grasp. So let's first think about how you would heapify a tree with just three elements.



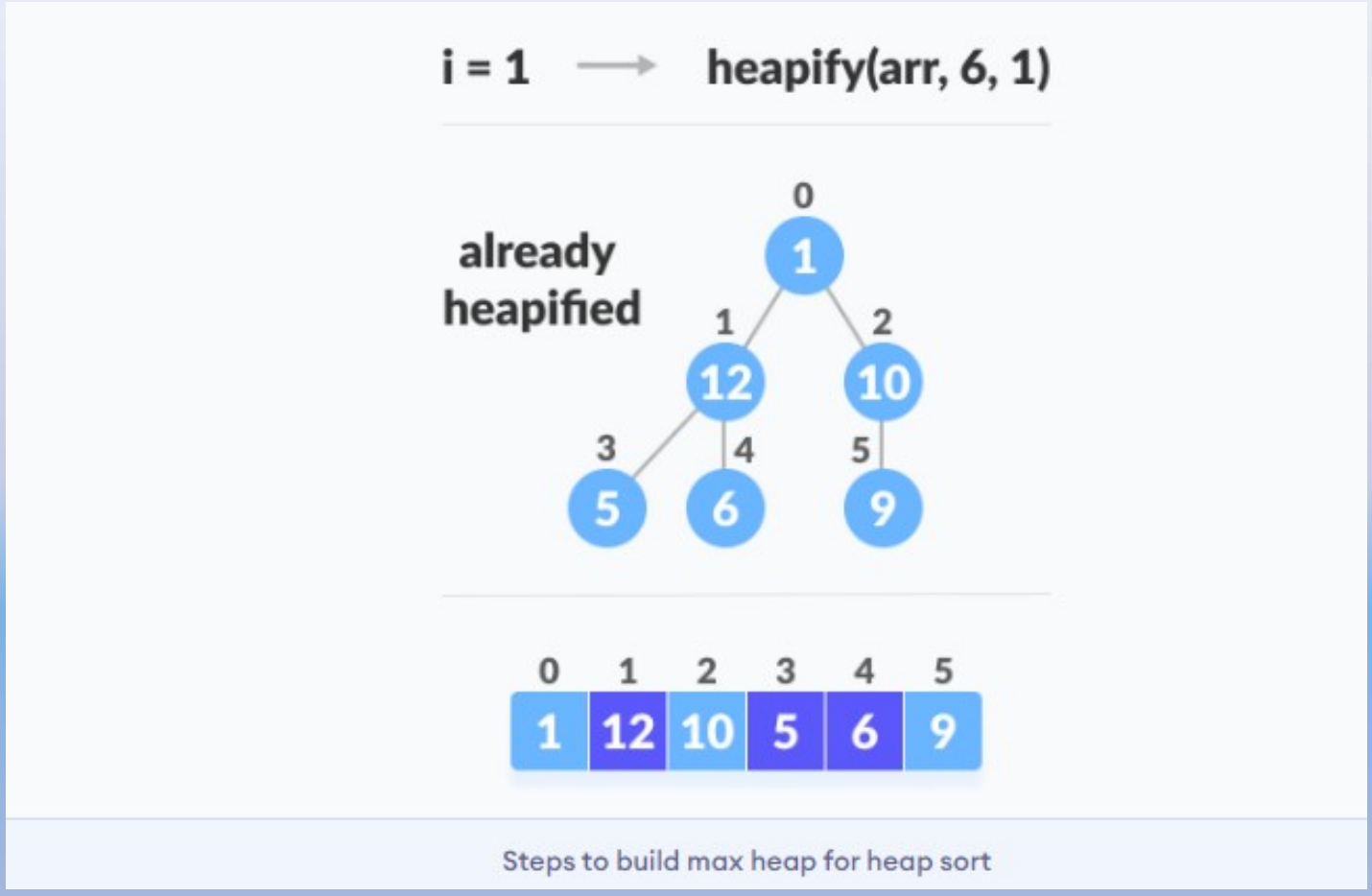
Heapify base cases



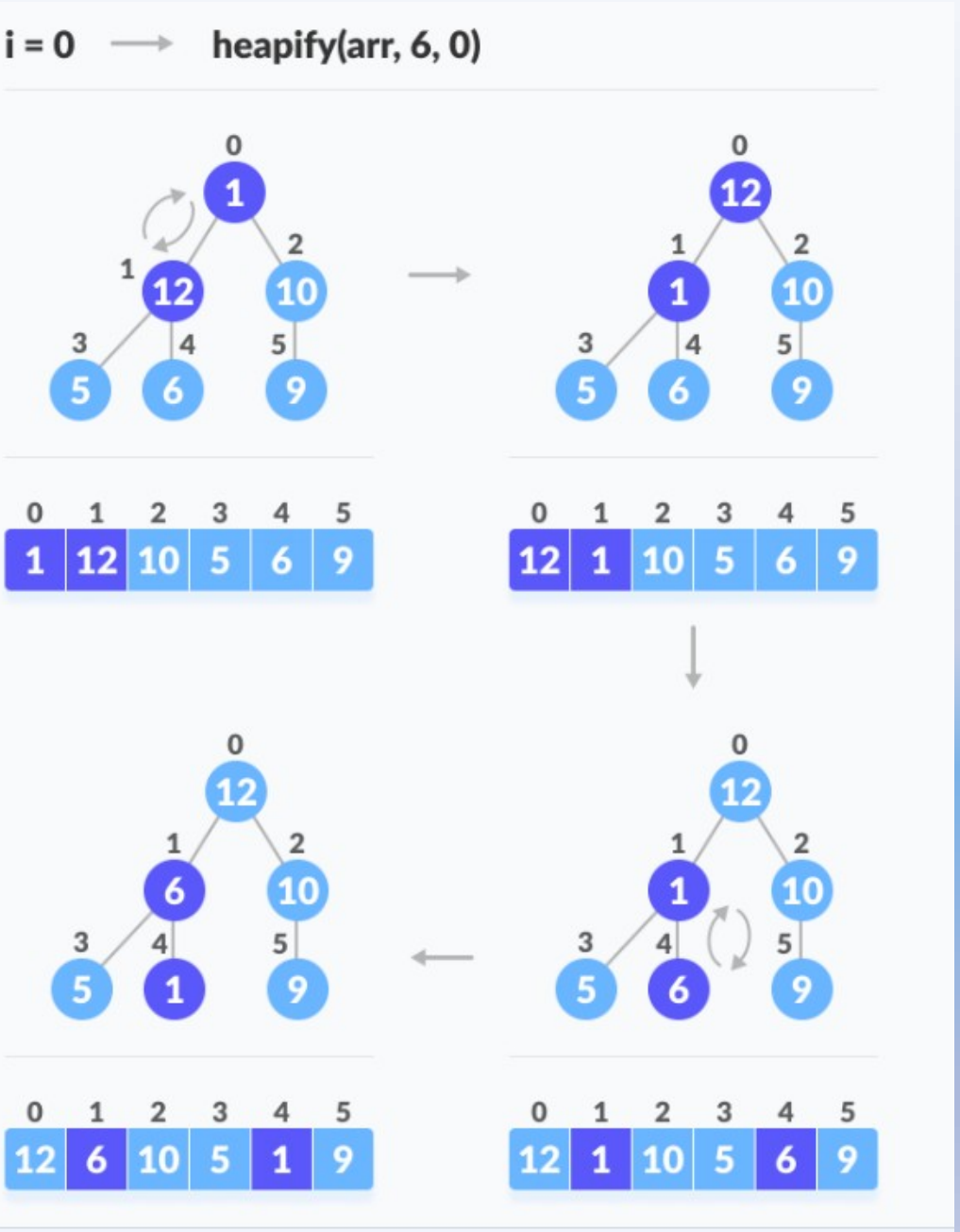
How to heapify root element when its subtrees are max-heaps



Steps to build max heap for heap sort



Steps to build max heap for heap sort



Steps to build max heap for heap sort

Working of Heap Sort

1. Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.
2. Swap: Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.
3. Remove: Reduce the size of the heap by 1.
4. Heapify: Heapify the root element again so that we have the highest element at root.
5. The process is repeated until all the items of the list are sorted.


Thank you !

ANY
QUESTION?