DATA STRUCTURES AND ITS APPLICATIONS

UE19CS202

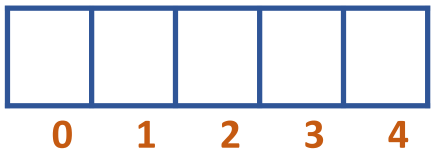
Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES AND ITS APPLICATIONS

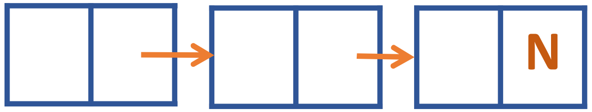
Basic Concept and Definitions: Trees

Shylaja S S
Department of Computer Science & Engineering

Linear Data Structures



List as an Array

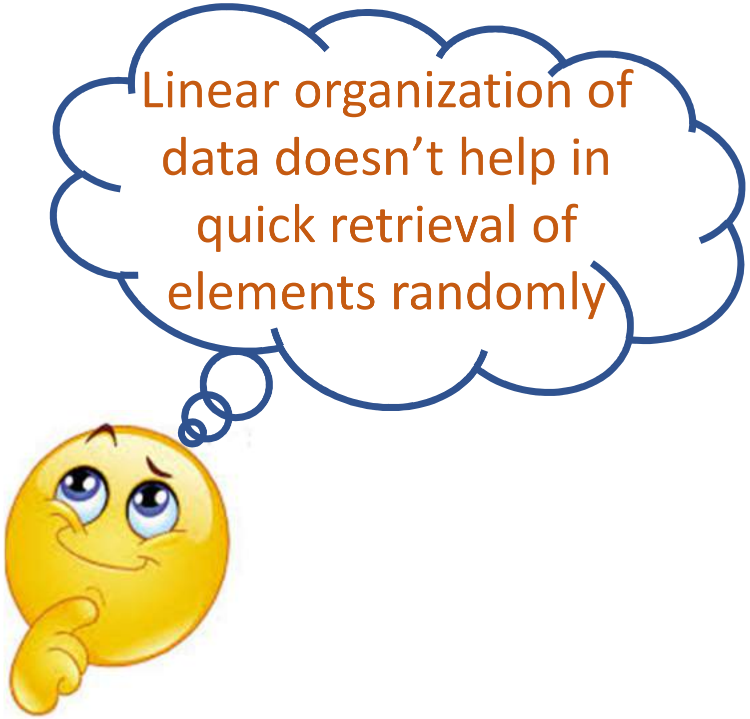


List as a Linked List

Disadvantage:
- Fixed Size
  - Expansion ✕
  - Shrink ✕
- Random Insertion & Deletion is Time Consuming

Disadvantage:
- Random Access is Time consuming

Introduction to Trees

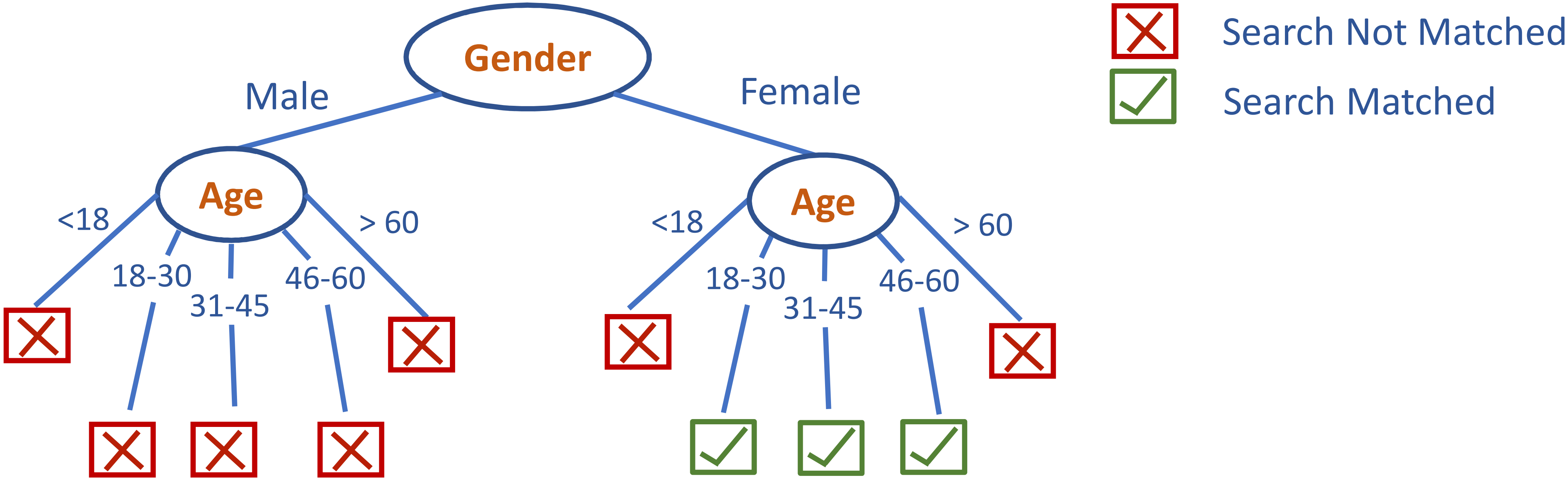


Example: To improve the probability of purchase of Women's Formal Wear in Less Time

Name: abc Gender: M Age: 25 email id: abc@xyz.com	**Name: def** **Gender: F** **Age: 21** **email id:** **def@xyz.com**	Name: ghi Gender: F Age: 10 email id: ghi@xyz.com	**…**	**Name: pqr** **Gender: F** **Age: 60** **email id:** **pqr@xyz.com**
0	**1**	**2**	**…**	**9999**

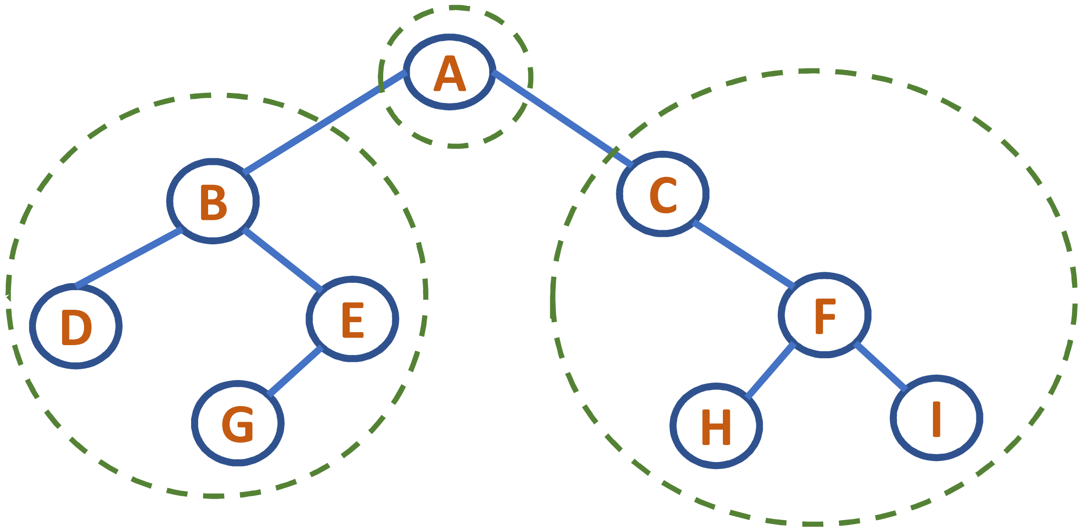
	…	**…**		**…**
0	**1**	**2**	**…**	**9999**

Example: To improve the probability of purchase of Women's Formal Wear in Less Time



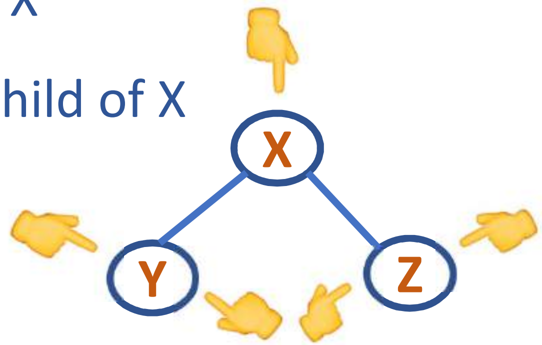
Binary Trees

- Non Linear Data Structure
- Finite set of elements that is either empty or is partitioned into three subsets
- First subset: is a single element, called the root
- Second subset: is a binary tree, called the left binary tree
- Third subset: is a binary tree, called the right binary tree



Binary Trees: Terminologies

- Each element of a binary tree is called a node of the tree
- Left node Y of X is called left child of X
- Right node Z of X is called the right child of X
- X is called the parent of Y and Z
- Y and Z are called siblings

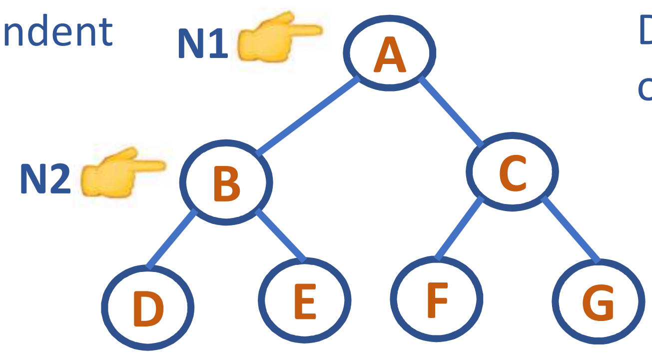


- A node which has no children is called leaf node/external node
- A node which has a child is called the non leaf node/internal node

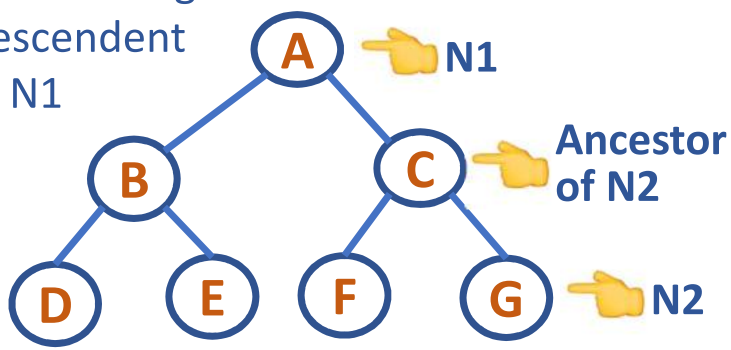
Binary Trees: Terminologies

- A node N1 is called the ancestor of a node N2 if
  - N1 is either the parent of N2 or
  - N1 is the parent of some ancestor of N2
- A node N2 becomes the descendent of node N1
- Descendent can be either the left descendent or the right descendent

N2 is the Left Descendent of N1



N2 is the Right Descendent of N1



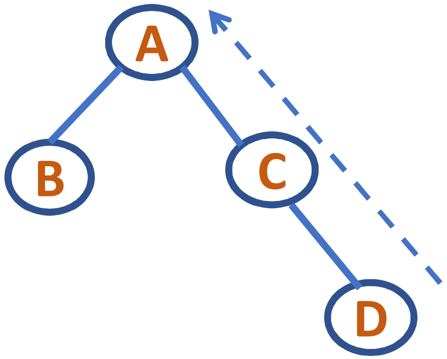
Binary Trees: Terminologies

- Level of a node

  Root has level 0; level of any other node is one more than its parent
- Depth of a tree

  Maximum level of any leaf in the tree (path length from the deepest leaf to the root)
- Depth of a node

  Path length from the node to the root

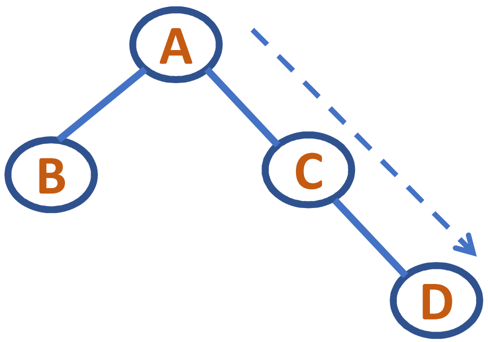


Level of node A – 0
Level of node B – 1
Level of node C – 1
Level of node D – 2

Depth of tree: 2
Depth of node A: 0
Depth of node B: 1
Depth of node C: 1
Depth of node D: 2

Binary Trees: Terminologies

- Height of a tree: Path length from the root node to the deepest leaf
- Height of a node: Path length from the node to the deepest leaf

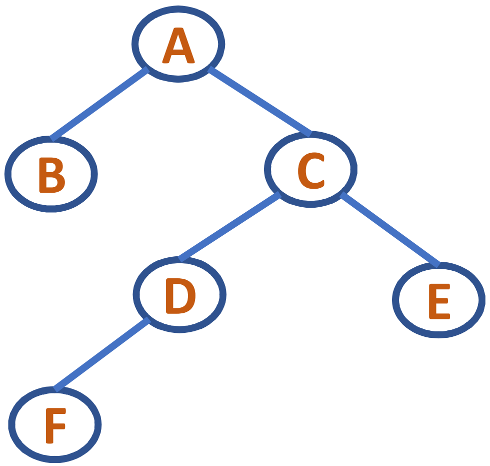


Height of Tree: 2

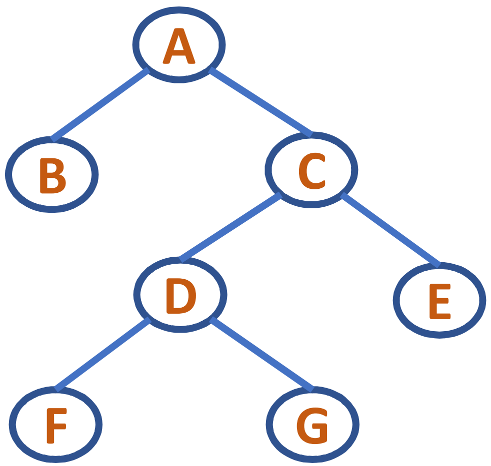
Height of Node A : 2
Height of Node B : 0
Height of Node C : 1
Height of Node D : 0

Strictly Binary Tree

A Binary tree where every node has either zero/two children



Not a Strictly Binary Tree

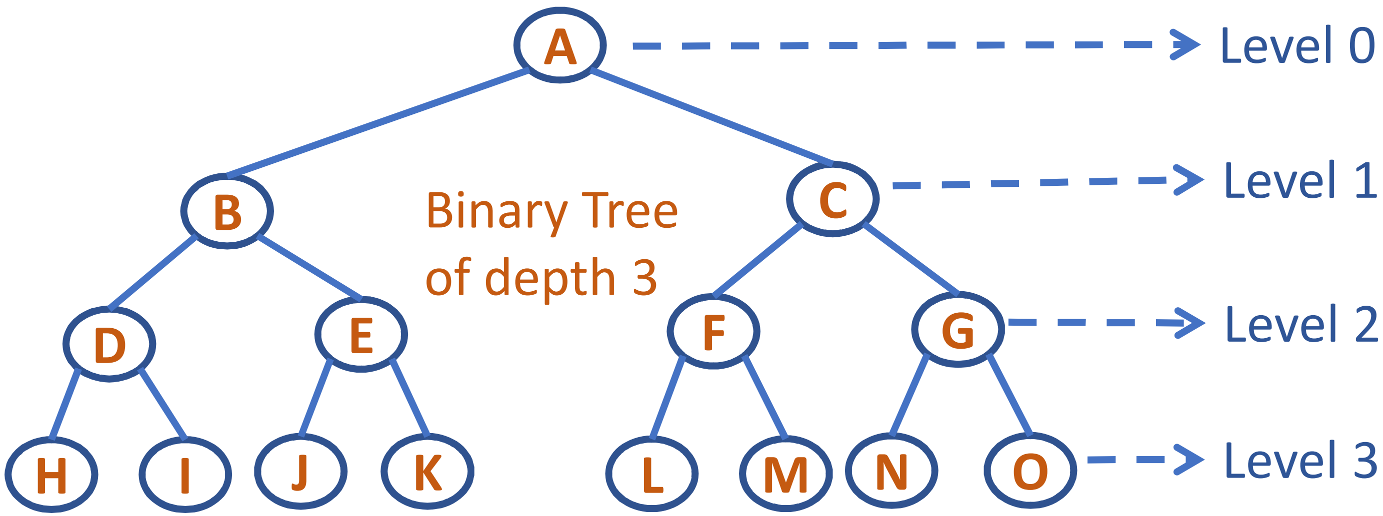


Strictly Binary Tree

Binary Trees: Terminologies

Fully Binary Tree

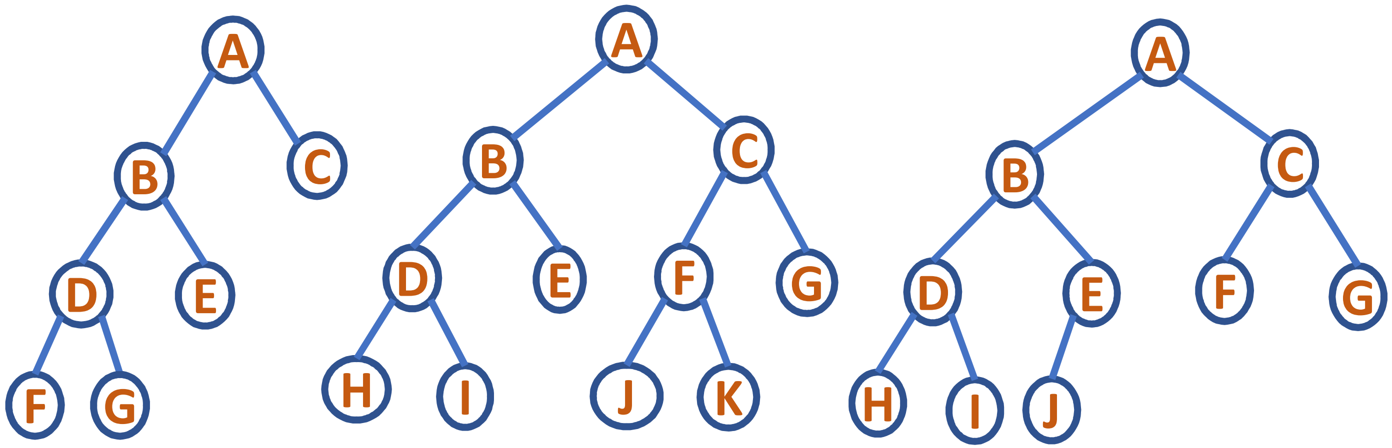
- A binary tree with all the leaves at the same level
- If the binary tree has depth d, then there are 0 to d levels
- Total no. of nodes = $2^0 + 2^1 + \ldots + 2^d = 2^{(d+1)} - 1$



Complete Binary Tree

For a Complete Binary Tree with n nodes and depth d:

- Any node nd at level less than d-1 has two children
- For any node nd of the tree with a right descendent at level d, nd must have a left child and every left descendent of nd is either a leaf at level d or has two children



Not Complete Binary Trees Complete Binary Tree

Binary Tree Properties

- Every node except the root has exactly one parent
- A tree with n nodes has n-1 edges (every node except the root has an edge to its parent)
- A tree consisting of only root node has height of zero
- The total number of nodes in a full binary tree of depth d is $2^{(d+1)} - 1$, $d \geq 0$
- For any non-empty binary tree, if n_0 is the number of leaf nodes and n_2 the nodes of degree 2, then $n_0 = n_2 + 1$

THANK YOU

Shylaja S S

Department of Computer Science
& Engineering

shylaja.sharath@pes.edu

+91 9449867804

DATA STRUCTURES AND ITS APPLICATIONS
UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

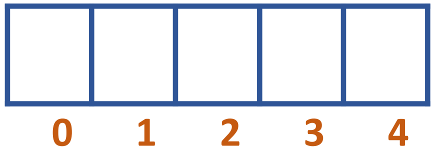
DATA STRUCTURES AND ITS APPLICATIONS

Basic Concept and Definitions: Trees

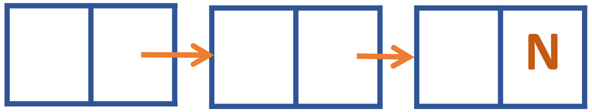
Shylaja S S

Department of Computer Science & Engineering

Linear Data Structures



List as an Array



List as a Linked List

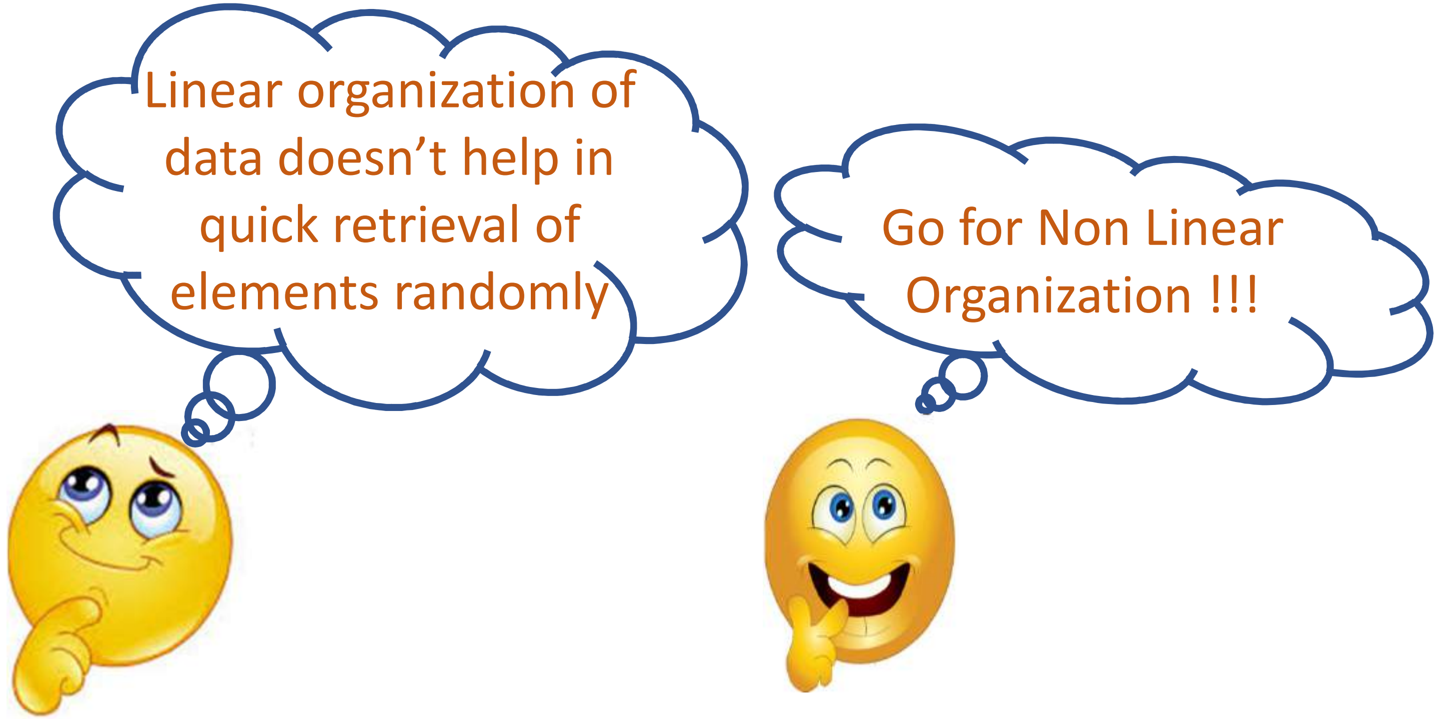
Disadvantage:

- Fixed Size
  - Expansion ✗
  - Shrink ✗
- Random Insertion & Deletion is Time Consuming

Disadvantage:

- Random Access is Time consuming

Introduction to Trees



Introduction to Trees

Example: To improve the probability of purchase of Women's Formal Wear in Less Time

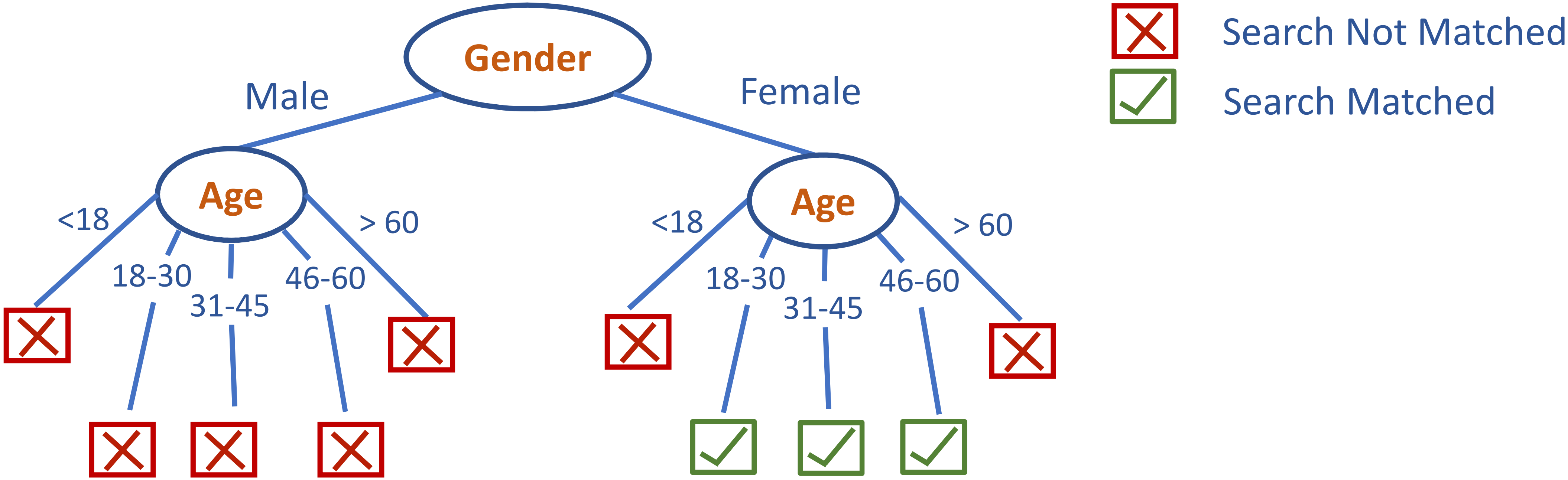
Name: abc Gender: M Age: 25 email id: abc@xyz.com	**Name: def** **Gender: F** **Age: 21** **email id:** **def@xyz.com**	Name: ghi Gender: F Age: 10 email id: ghi@xyz.com	…	**Name: pqr** **Gender: F** **Age: 60** **email id:** **pqr@xyz.com**
0	1	2	…	9999

	…	…		…
0	1	2	…	9999

DATA STRUCTURES AND ITS APPLICATIONS

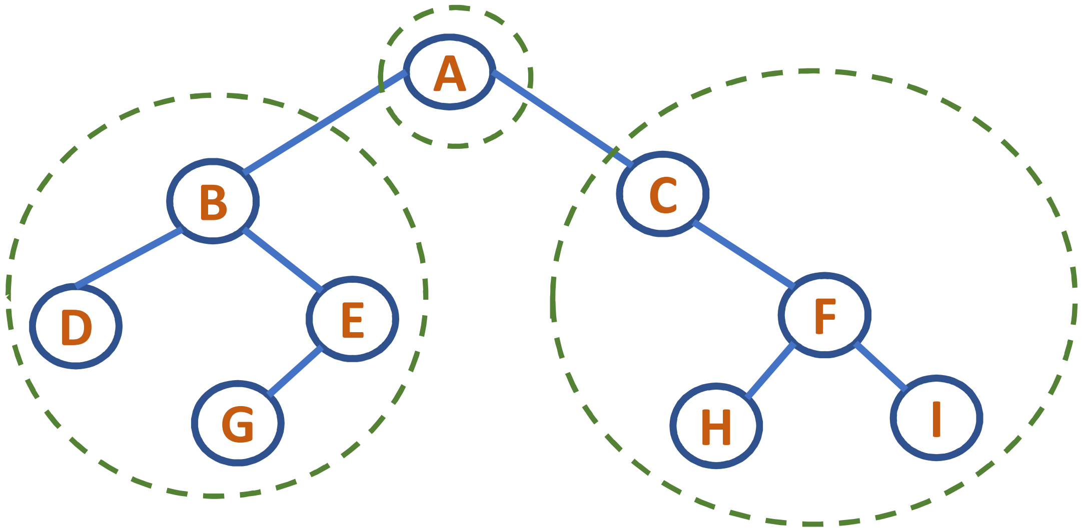
Introduction to Trees

Example: To improve the probability of purchase of Women's Formal Wear in Less Time



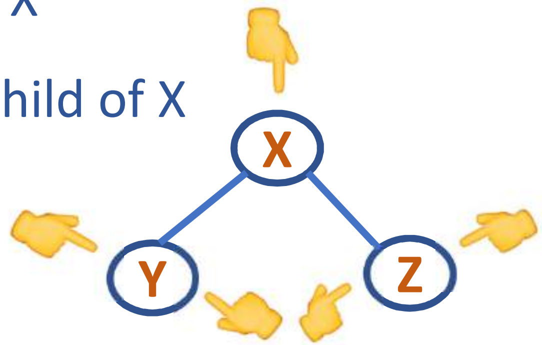
Binary Trees

- Non Linear Data Structure
- Finite set of elements that is either empty or is partitioned into three subsets
- First subset: is a single element, called the root
- Second subset: is a binary tree, called the left binary tree
- Third subset: is a binary tree, called the right binary tree



Binary Trees: Terminologies

- Each element of a binary tree is called a node of the tree
- Left node Y of X is called left child of X
- Right node Z of X is called the right child of X
- X is called the parent of Y and Z
- Y and Z are called siblings

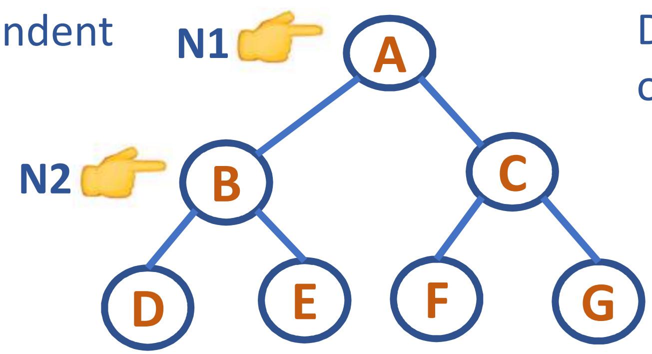


- A node which has no children is called leaf node/external node
- A node which has a child is called the non leaf node/internal node

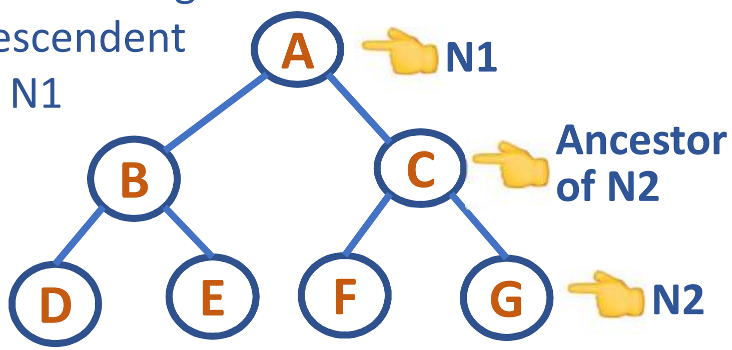
Binary Trees: Terminologies

- A node N1 is called the ancestor of a node N2 if
  - N1 is either the parent of N2 or
  - N1 is the parent of some ancestor of N2
- A node N2 becomes the descendent of node N1
- Descendent can be either the left descendent or the right descendent

N2 is the Left Descendent of N1



N2 is the Right Descendent of N1



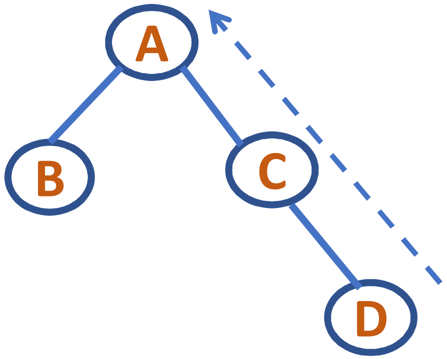
Binary Trees: Terminologies

- Level of a node

  Root has level 0; level of any other node is one more than its parent
- Depth of a tree

  Maximum level of any leaf in the tree (path length from the deepest leaf to the root)
- Depth of a node

  Path length from the node to the root

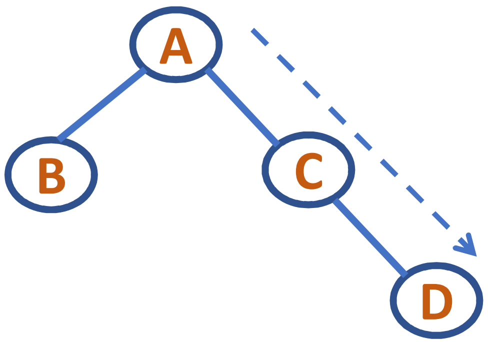


Level of node A – 0
Level of node B – 1
Level of node C – 1
Level of node D – 2

Depth of tree: 2
Depth of node A: 0
Depth of node B: 1
Depth of node C: 1
Depth of node D: 2

Binary Trees: Terminologies

- Height of a tree: Path length from the root node to the deepest leaf
- Height of a node: Path length from the node to the deepest leaf

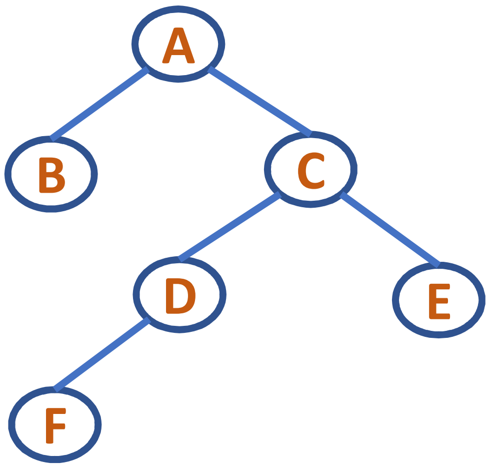


Height of Tree: 2

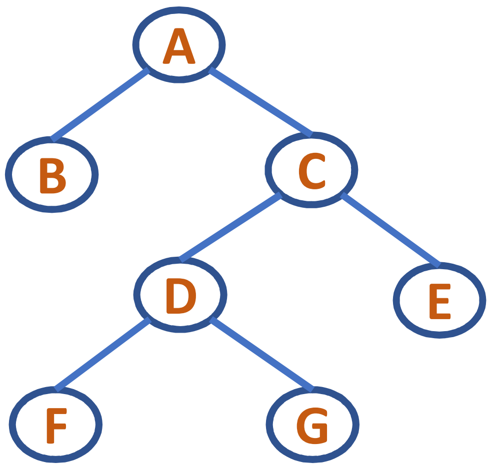
Height of Node A : 2
Height of Node B : 0
Height of Node C : 1
Height of Node D : 0

Strictly Binary Tree

A Binary tree where every node has either zero/two children



Not a Strictly Binary Tree

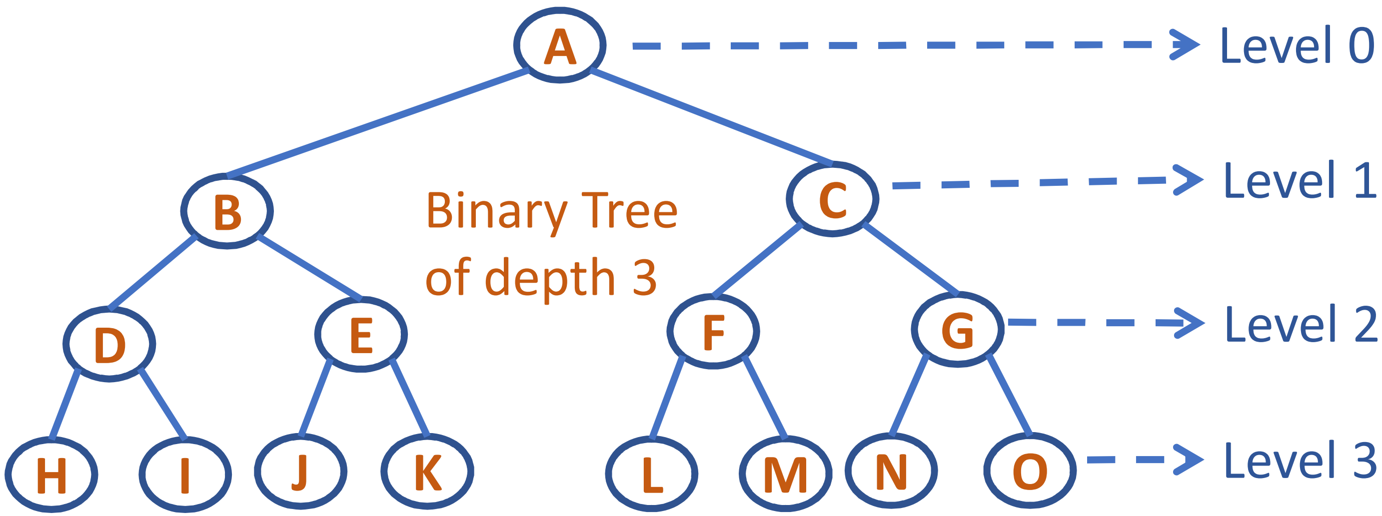


Strictly Binary Tree

Binary Trees: Terminologies

Fully Binary Tree

- A binary tree with all the leaves at the same level
- If the binary tree has depth d, then there are 0 to d levels
- Total no. of nodes = $2^0 + 2^1 + \ldots + 2^d = 2^{(d+1)} - 1$

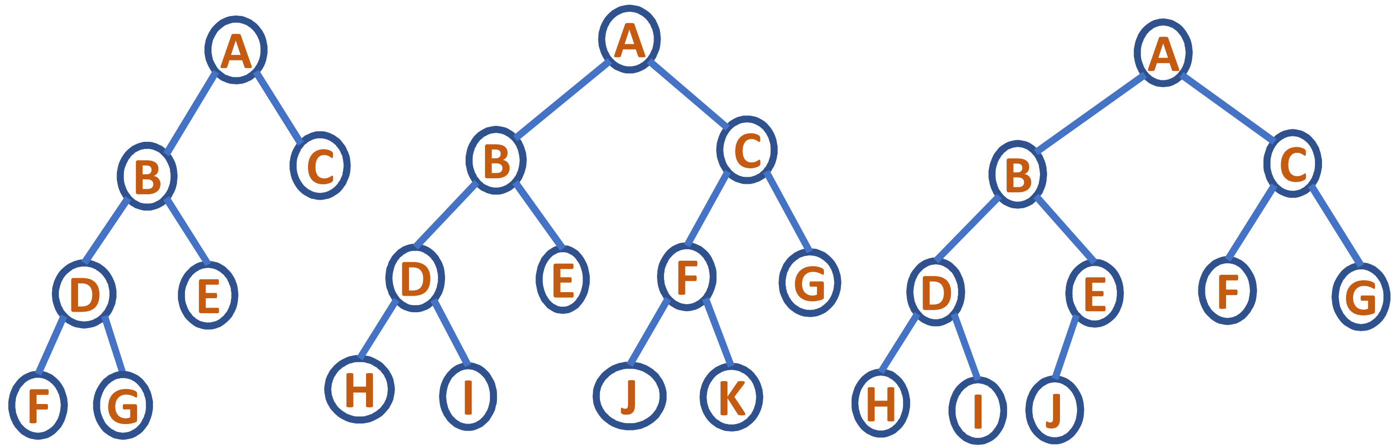


Complete Binary Tree

For a Complete Binary Tree with n nodes and depth d:

• Any node nd at level less than d-1 has two children

• For any node nd of the tree with a right descendent at level d, nd must have a left child and every left descendent of nd is either a leaf at level d or has two children



Not Complete Binary Trees   Complete Binary Tree

Binary Tree Properties

- Every node except the root has exactly one parent
- A tree with n nodes has n-1 edges (every node except the root has an edge to its parent)
- A tree consisting of only root node has height of zero
- The total number of nodes in a full binary tree of depth d is $2^{(d+1)} - 1$, $d \geq 0$
- For any non-empty binary tree, if n_0 is the number of leaf nodes and n_2 the nodes of degree 2, then $n_0 = n_2 + 1$

THANK YOU

Shylaja S S

Department of Computer Science & Engineering

shylaja.sharath@pes.edu

+91 9449867804

DATA STRUCTURES AND ITS APPLICATIONS
UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES AND ITS APPLICATIONS

BST Implementation using Dynamic Allocation: Insertion

Shylaja S S
Department of Computer Science & Engineering

Background

Problem : find a target key in a list of elements

Sequential: Potentially enumerate every key

Ordered List: Searching can be done on logn

Frequent insertions and deletions : Ordered List is much slower

Solution: Binary Trees provide an excellent solution to this by organizing every element in the list as a node in the tree

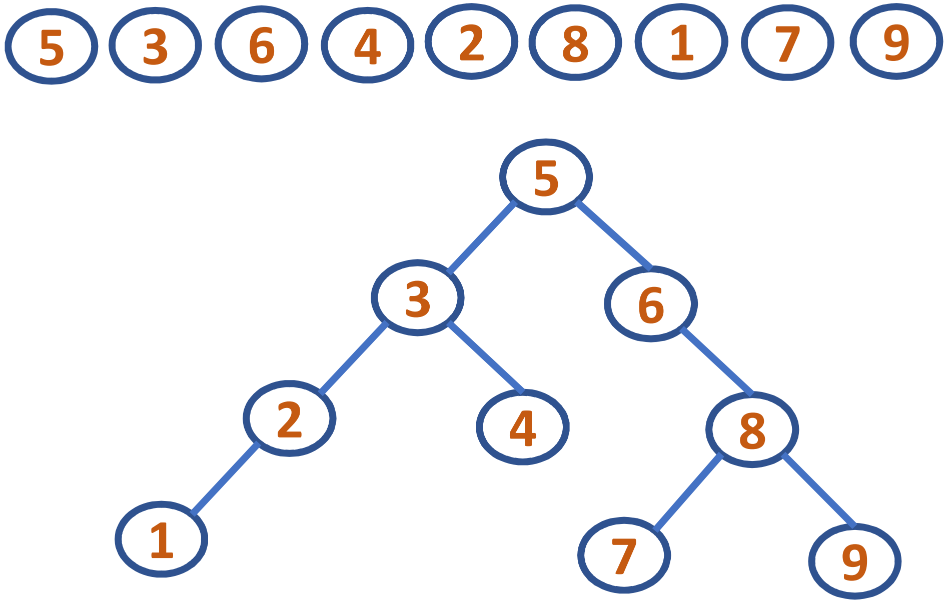
Binary Search Tree: Definition

A Binary Search Tree is a binary tree which has the following properties:

- all the elements in the left subtree of a node **n** are less than the contents of node **n**
- all the elements in the right subtree of a node **n** are greater than or equal to the contents of node **n**

Binary Search Tree – An Application of Binary Tree

A Binary Search Tree with the nodes inserted in the order:
5, 3, 6, 4, 2, 8, 1,7, 9



Linked implementation

Here every node will have its own **info** along with the **links to left child** and **right child**

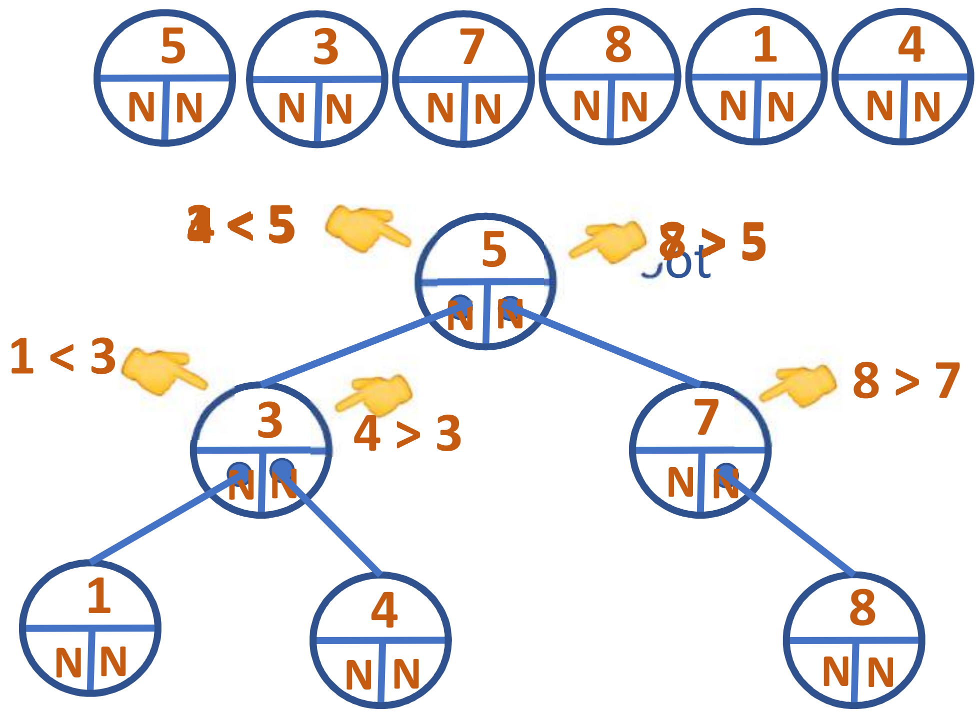
```
typedef struct tree_linked
{
  int info;
  struct tree_linked *left,*right;
}NODE;

NODE *root=NULL; //root points to Root of the tree and initially it is null
```

DATA STRUCTURES AND ITS APPLICATIONS

Binary Search Tree - Implementation

Linked implementation: 5, 3, 7, 8, 1, 4



THANK YOU

Shylaja S S

Department of Computer Science
& Engineering

shylaja.sharath@pes.edu

+91 9449867804

DATA STRUCTURES AND ITS APPLICATIONS
UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES AND ITS APPLICATIONS

BST: Deletion Operations

Shylaja S S
Department of Computer Science & Engineering

Deletion of a Node in Binary Search Tree

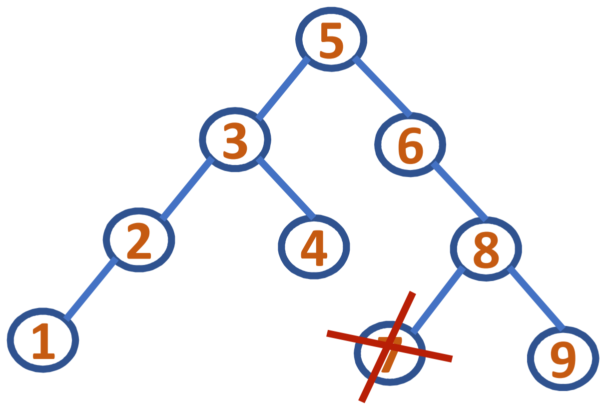
case1: Node with no child (leaf node)

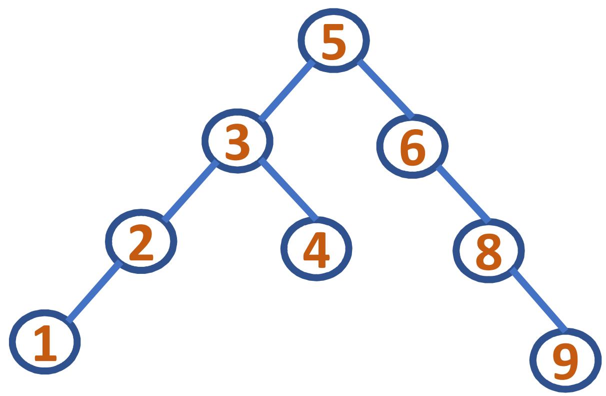
case2: Node with 1 child

case3: Node with 2 children

Binary Search Tree - Deletion

case1: Node with no child (leaf node)



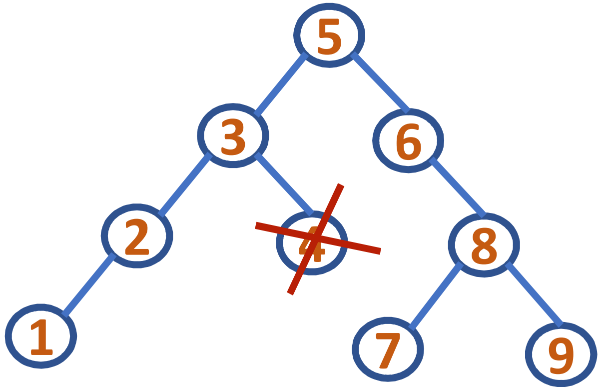


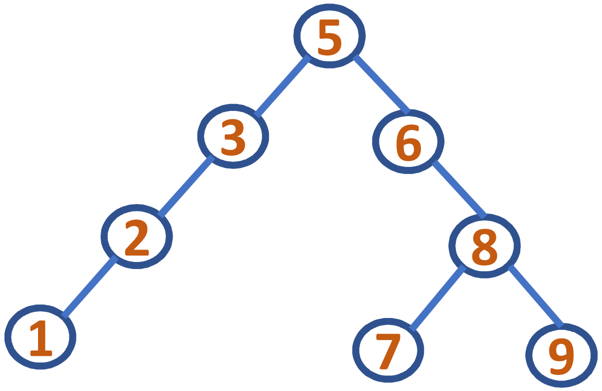
To delete the node with info 7:

- Set its parent's left child field to point to NULL
- Free memory allocated to node with info 7

Binary Search Tree - Deletion

case1: Node with no child (leaf node)



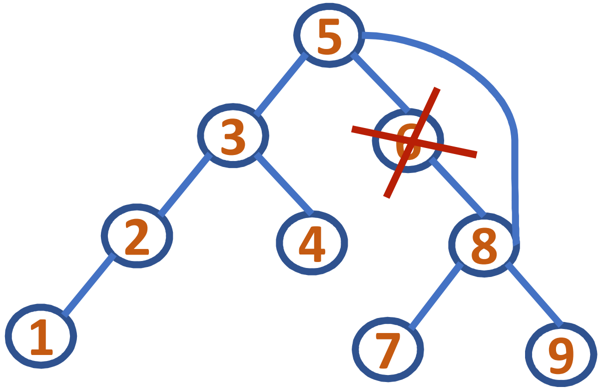


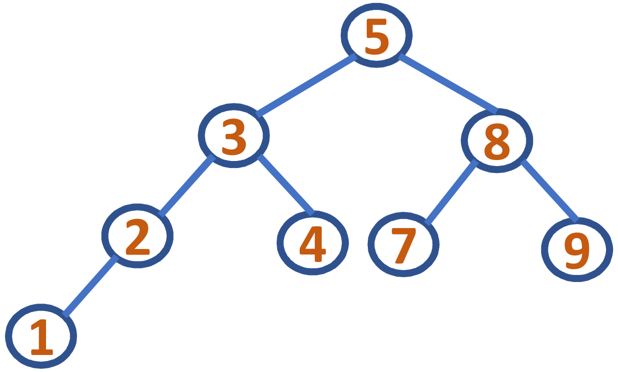
To delete the node with info 4:

- Set its parent's right child field to point to NULL
- Free memory allocated to node with info 4

Binary Search Tree - Deletion

case2: Node with 1 child



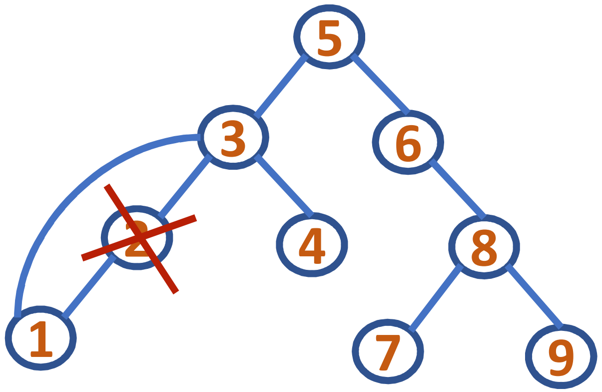


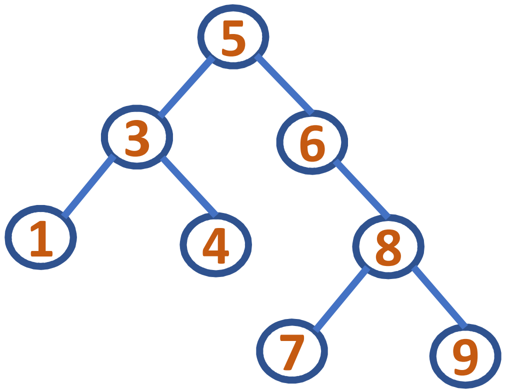
To delete the node with info 6:

- Set its parent's right child field to point to its only child
- Free memory allocated to node with info 6

Binary Search Tree - Deletion

case2: Node with 1 child





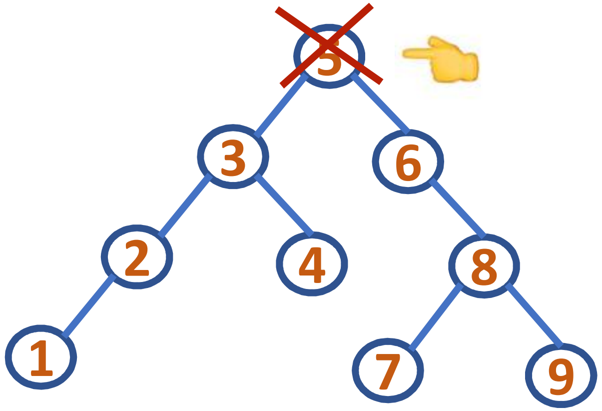
To delete the node with info 2:

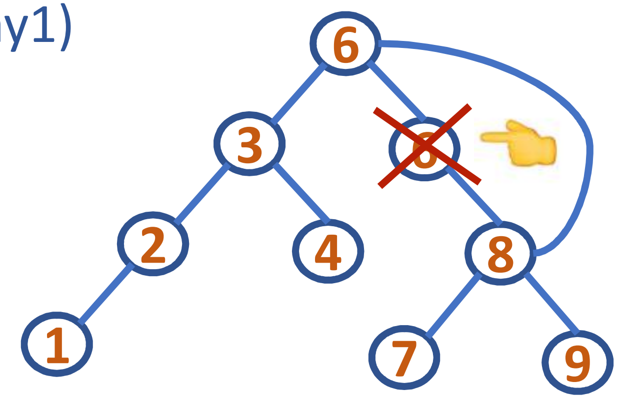
- Set its parent's left child field to point to its only child
- Free memory allocated to node with info 2

Binary Search Tree - Deletion

case3: Node with 2 children(Replace with inorder successor)

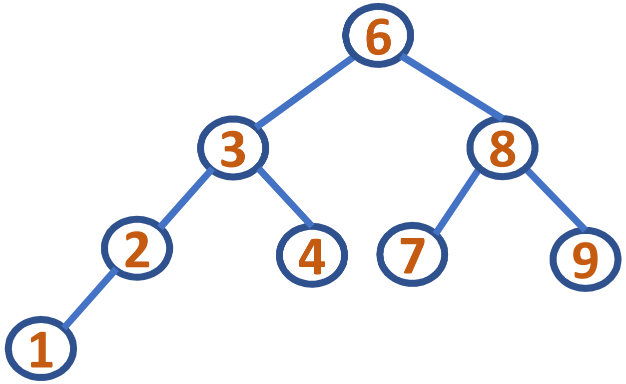
(Way1)





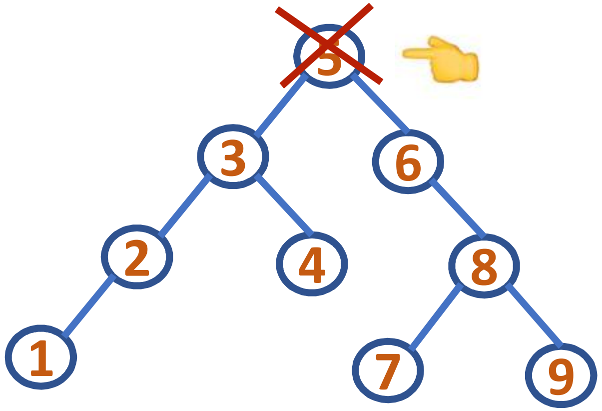
To delete the node with info 5:

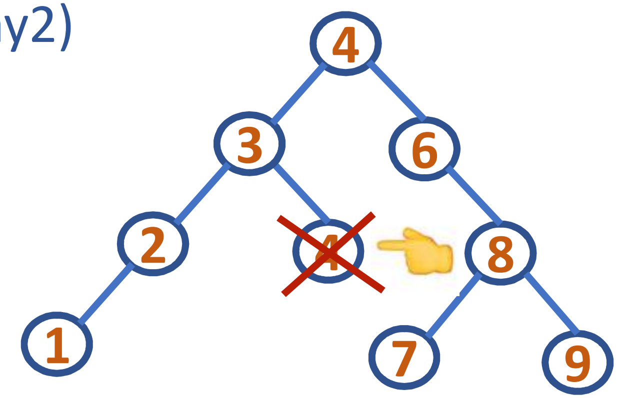
- Replace 5 with its **inorder successor** and delete that inorder successor
- Now case3 has got changed to case2 (In general may change to case2 or case1)



Binary Search Tree - Deletion

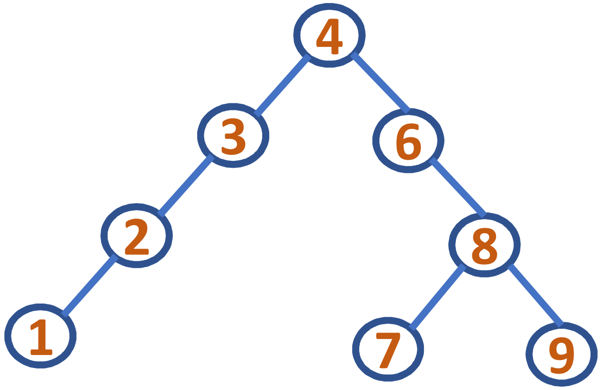
case3: Node with 2 children(Replace with inorder predecessor)
(Way2)





To delete the node with info 5:

- Replace 5 with its **inorder predecessor** and delete that inorder predecessor
- Here case3 has got changed to case1 (In general may change to case2 or case1)



THANK YOU

Shylaja S S

Department of Computer Science
& Engineering

shylaja.sharath@pes.edu

+91 9449867804

DATA STRUCTURES AND ITS APPLICATIONS
UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

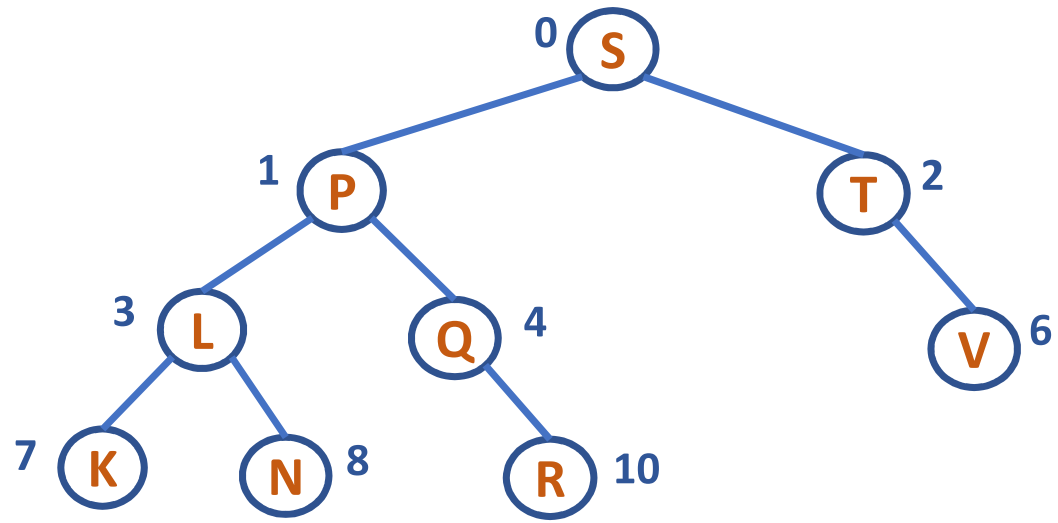
DATA STRUCTURES AND ITS APPLICATIONS

BST: Implementation using Arrays

Shylaja S S
Department of Computer Science & Engineering

Binary Search Tree - Implementation

Array Implementation (Implicit implementation)



S	P	T	L	Q	…	V	K	N	…	R
0	1	2	3	4	5	6	7	8	9	10

Binary Search Tree - Implementation

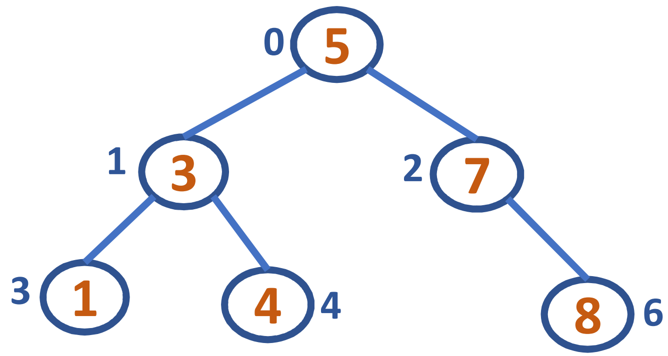
Array Implementation (Implicit implementation)

```
typedef struct tree_array
{
        int info;
        int used;
}NODE;
```

- NODE bst[MAX]; //here bst is an array of nodes
- each node has its data and another field by name used to contain whether it is a valid node or not
- used = 1 or 0

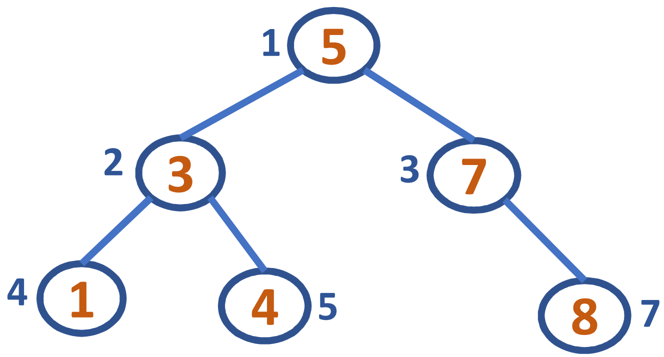
Binary Search Tree - Implementation

Array Implementation: 5, 3, 7, 8, 1, 4



Root Position: i = 0
Left Child Position: 2i + 1
Right Child Position: 2i + 2

OR



Root Position: i = 1
Left Child Position: 2i
Right Child Position: 2i + 1

info	5	3	7	1	4		8
used	1	1	1	1	1	0	1
Position: i	0	1	2	3	4	5	6

info		5	3	7	1	4		8
used	0	1	1	1	1	1	0	1
Position: i	0	1	2	3	4	5	6	7

Binary Search Tree - Implementation

Array Implementation: 5, 3, 7, 8, 1, 4

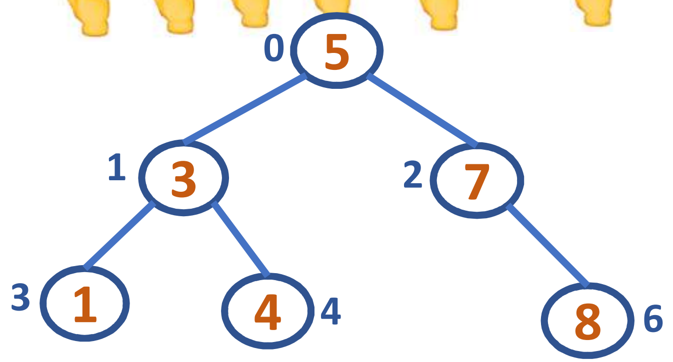
Root Position: i = 0
Left Child Position: 2i + 1
Right Child Position: 2i + 2

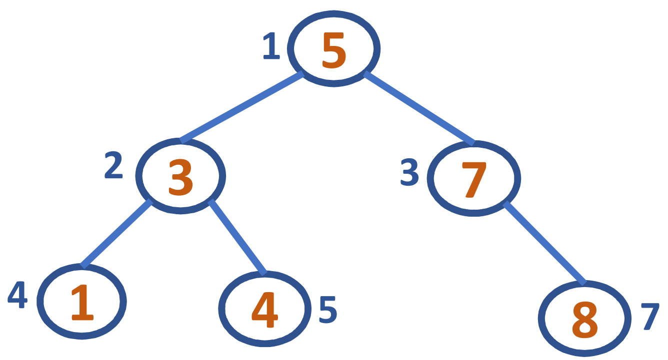
OR

Root Position: i = 1
Left Child Position: 2i
Right Child Position: 2i + 1

info	5	3	7	1	4		8
used	1	1	1	1	1	0	1
Position: i	0	1	2	3	4	5	6

info		5	3	7	1	4		8
used	0	1	1	1	1	1	0	1
Position: i	0	1	2	3	4	5	6	7





THANK YOU

Shylaja S S

Department of Computer Science
& Engineering

shylaja.sharath@pes.edu

+91 9449867804

DATA STRUCTURES AND ITS APPLICATIONS
UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Binary Tree Traversal

Shylaja S S

Department of Computer Science & Engineering

Binary Tree Traversals

Important operation: Traversal

Traversal: Moving through all the nodes in a binary tree and visiting each one in turn

Trees: There are many orders possible since it is a nonlinear DS

Tasks: 1. Visiting a node denoted by V

2. Traversing the left subtree denoted by L

3. Traversing the right subtree denoted by R

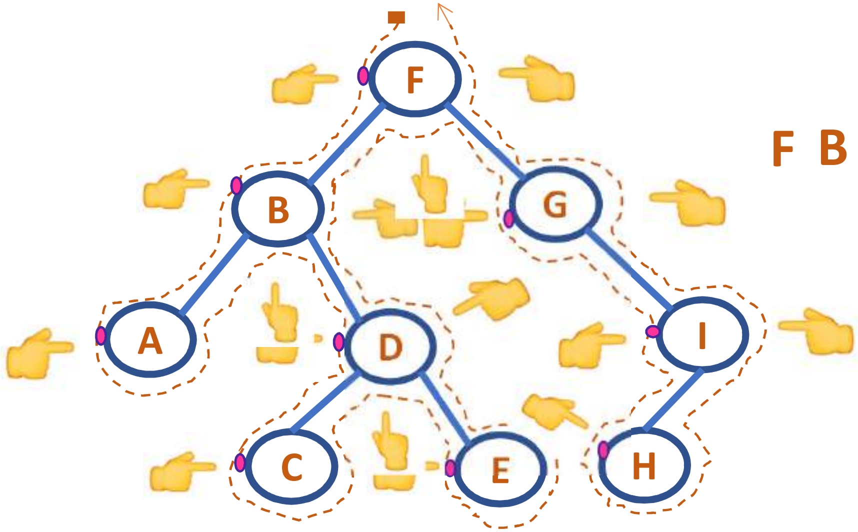
Six ways to arrange them: VLR, LVR, LRV, VRL, RVL, RLV

Standard Traversals include: VLR-Preorder, LVR-Inorder, LRV-Postorder

DATA STRUCTURES AND ITS APPLICATIONS
Binary Tree Traversal: Preorder

Steps:

- Root Node is visited before the subtrees
- Left subtree is traversed in preorder
- Right subtree is traversed in preorder

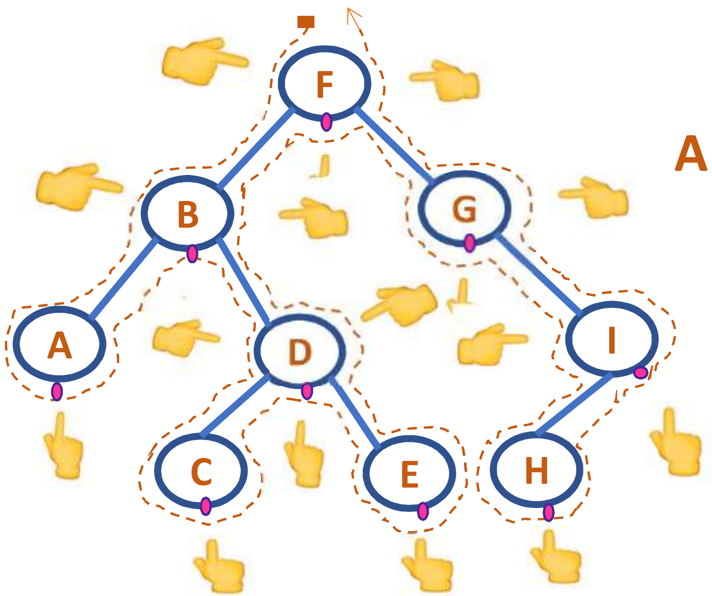


F B A D C E G I H

Binary Tree Traversal: Inorder

Steps:

- Left subtree is traversed in Inorder
- Root Node is visited
- Right subtree is traversed in Inorder

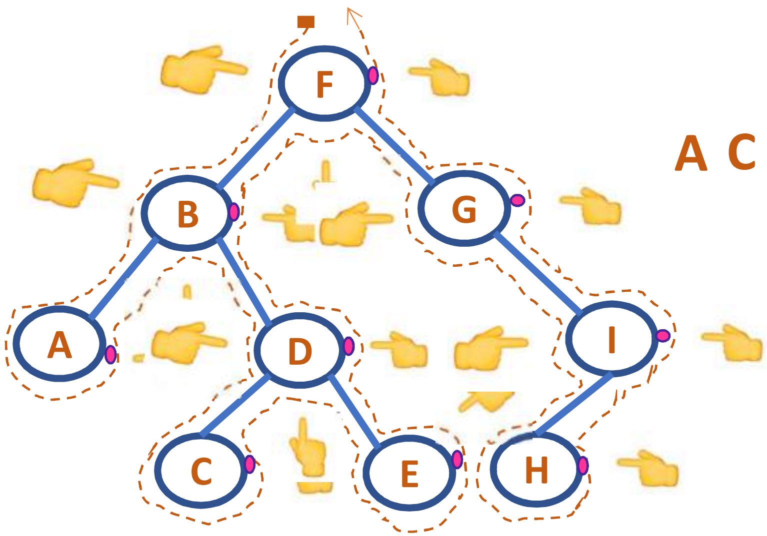


A B C D E F G H I

Binary Tree Traversal: Postorder

Steps:

- Left subtree is traversed in postorder
- Right subtree is traversed in postorder
- Root Node is visited



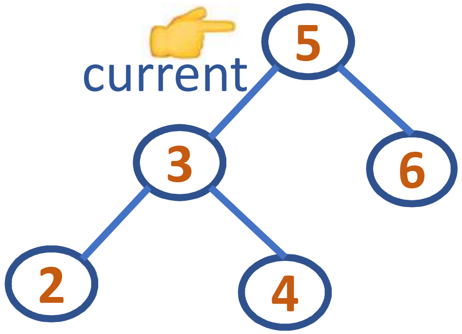
A C E D B H I G F

Iterative Inorder Traversal

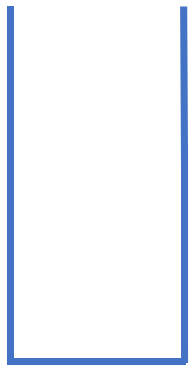
```
iterativeInorder(root)
 s = emptyStack
current = root
do {
        while(current != null)
        {       /* Travel down left branches as far as possible
                saving pointers to nodes passed in the stack*/
                push(s, current)
                current = current->left
        } //At this point, the left subtree is empty
        poppedNode = pop(s)
        print poppedNode ->info        //visit the node
        current = poppedNode ->right //traverse right subtree
} while(!isEmpty(s) or current != null)
```

Iterative Inorder Traversal

```
iterativeInorder(root)
 s = emptyStack
current = root
do {
      while(current != null)
     {
         push(s, current)
         current = current->left
      }
      poppedNode = pop(s)
      print poppedNode ->info
      current = poppedNode ->right
} while(!isEmpty(s) or current != null)
```



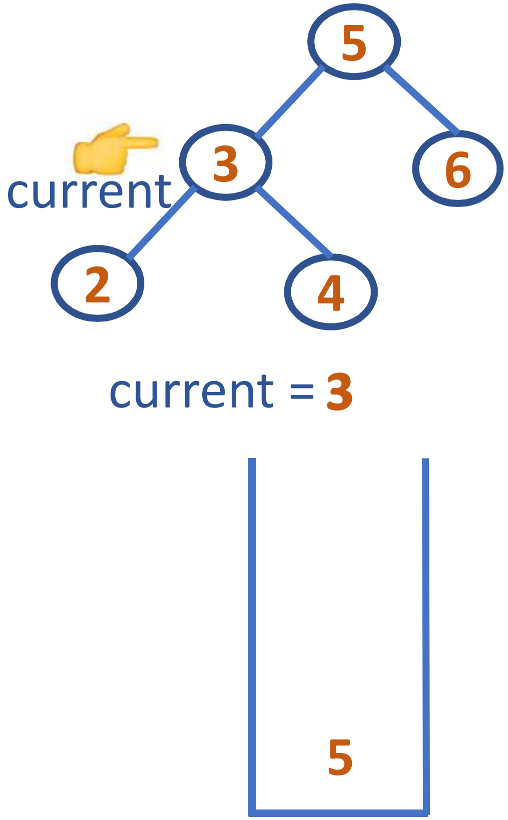
current = **5**



Note: Stack has Address of Nodes Pushed In

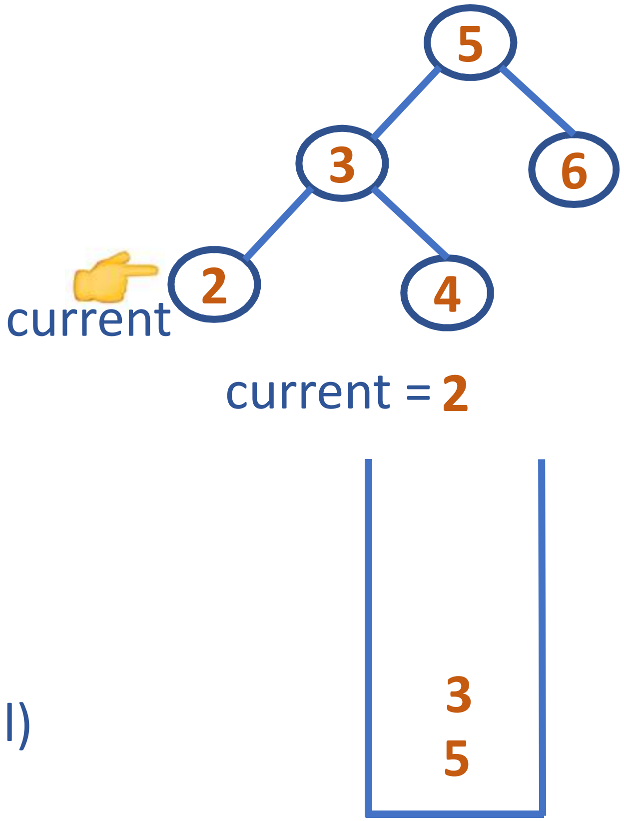
Iterative Inorder Traversal

```
iterativeInorder(root)
 s = emptyStack
current = root
do {
      while(current != null)
     {
          push(s, current)
          current = current->left
      }
      poppedNode = pop(s)
      print poppedNode ->info
      current = poppedNode ->right
} while(!isEmpty(s) or current != null)
```



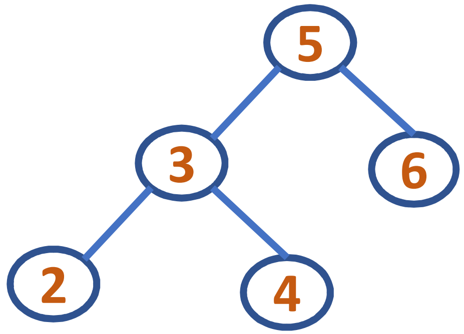
Iterative Inorder Traversal

```
iterativeInorder(root)
 s = emptyStack
current = root
do {
      while(current != null)
     {
          push(s, current)
          current = current->left
      }
      poppedNode = pop(s)
      print poppedNode ->info
      current = poppedNode ->right
} while(!isEmpty(s) or current != null)
```

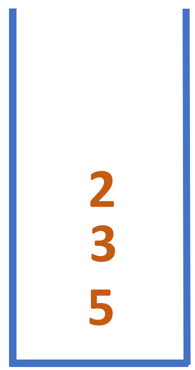


Iterative Inorder Traversal

```
iterativeInorder(root)
 s = emptyStack
current = root
do {
      while(current != null)
     {
          push(s, current)
          current = current->left
      }
      poppedNode = pop(s)
      print poppedNode ->info
      current = poppedNode ->right
} while(!isEmpty(s) or current != null)
```

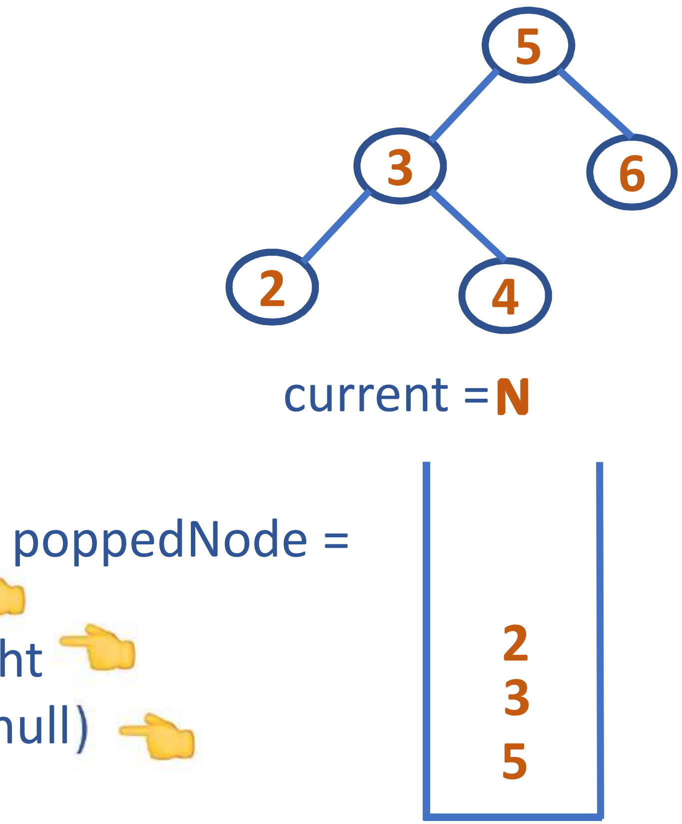


current = **N**



Iterative Inorder Traversal

```
iterativeInorder(root)
 s = emptyStack
current = root
do {
      while(current != null)
     {
          push(s, current)
          current = current->left
      }
      poppedNode = pop(s)
      print poppedNode ->info
      current = poppedNode ->right
} while(!isEmpty(s) or current != null)
```

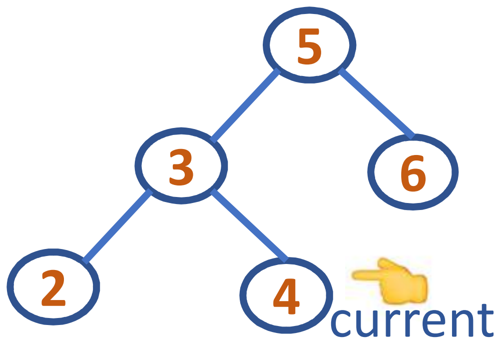


Inorder Traversal:

2

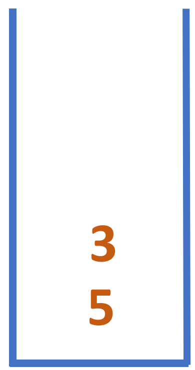
Iterative Inorder Traversal

```
iterativeInorder(root)
 s = emptyStack
current = root
do {
      while(current != null)
     {
          push(s, current)
          current = current->left
      }
      poppedNode = pop(s)
      print poppedNode ->info
      current = poppedNode ->right
} while(!isEmpty(s) or current != null)
```



current = N

poppedNode = 2



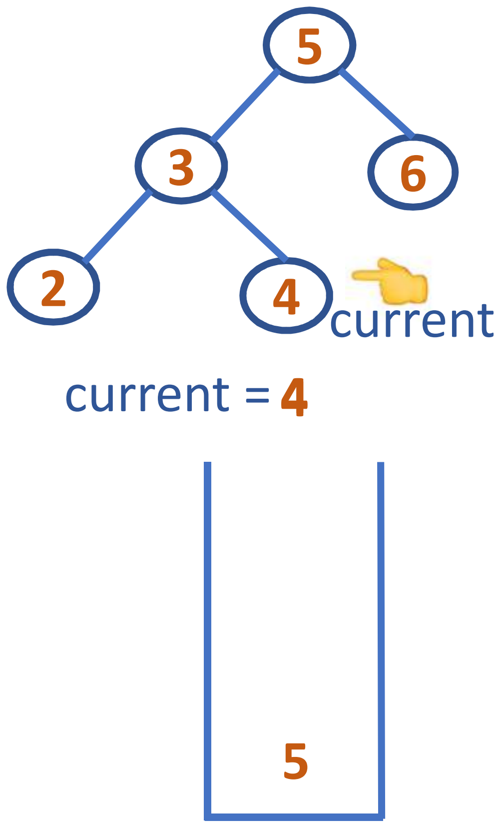
Inorder Traversal:

2 3

DATA STRUCTURES AND ITS APPLICATIONS

Iterative Inorder Traversal

```
iterativeInorder(root)
 s = emptyStack
current = root
do {
      while(current != null)
      {
          push(s, current)
          current = current->left
      }
      poppedNode = pop(s)
      print poppedNode ->info
      current = poppedNode ->right
} while(!isEmpty(s) or current != null)
```

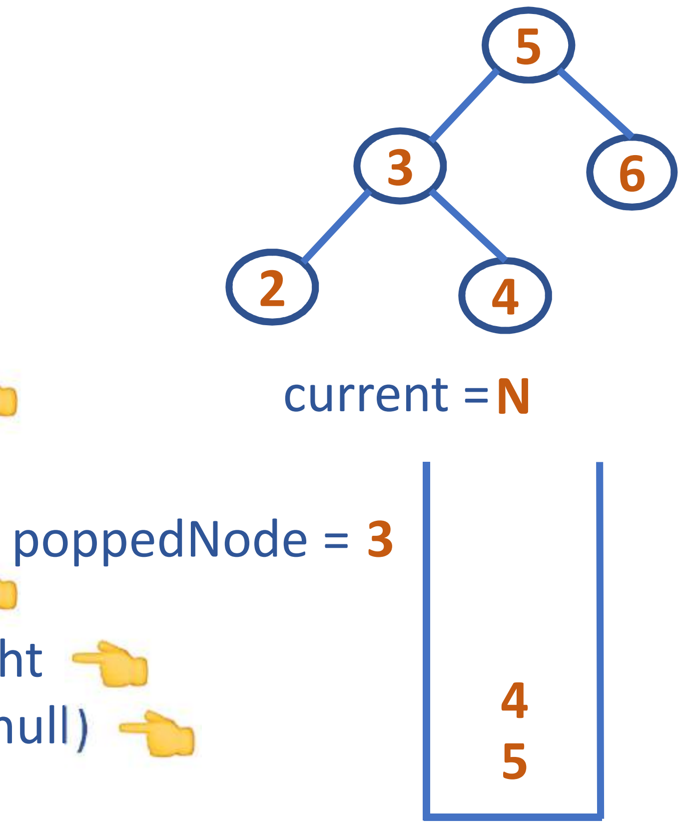


Inorder Traversal:

2 3

Iterative Inorder Traversal

```
iterativeInorder(root)
 s = emptyStack
current = root
do {
      while(current != null)
     {
          push(s, current)
          current = current->left
      }
      poppedNode = pop(s)
      print poppedNode ->info
      current = poppedNode ->right
} while(!isEmpty(s) or current != null)
```

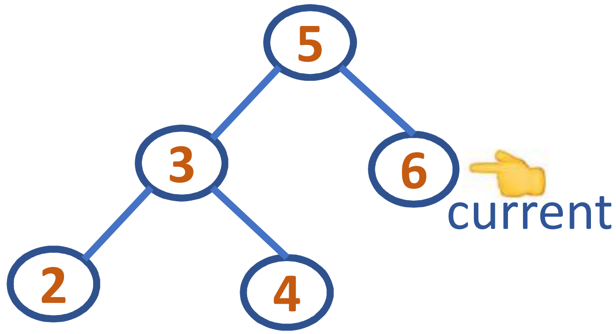


Inorder Traversal:

2 3 4

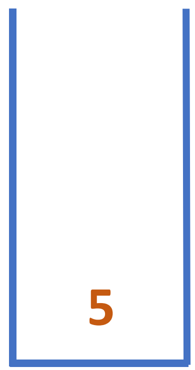
Iterative Inorder Traversal

```
iterativeInorder(root)
 s = emptyStack
current = root
do {
      while(current != null)
     {
          push(s, current)
          current = current->left
      }
      poppedNode = pop(s)
      print poppedNode ->info
      current = poppedNode ->right
} while(!isEmpty(s) or current != null)
```



current = 6

poppedNode = 4

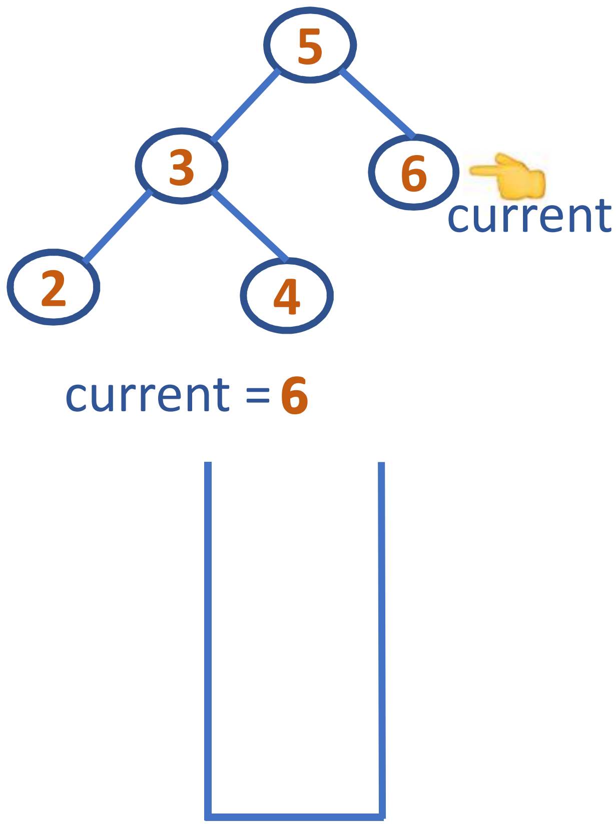


Inorder Traversal:

2 3 4 5

Iterative Inorder Traversal

```
iterativeInorder(root)
 s = emptyStack
current = root
do {
      while(current != null)
     {
           push(s, current)
           current = current->left
      }
      poppedNode = pop(s)
      print poppedNode ->info
      current = poppedNode ->right
} while(!isEmpty(s) or current != null)
```

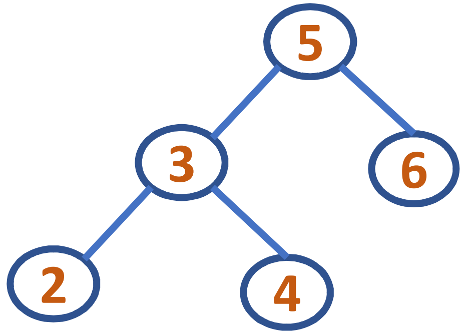


Inorder Traversal:

2 3 4 5

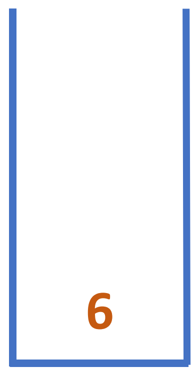
Iterative Inorder Traversal

```
iterativeInorder(root)
 s = emptyStack
current = root
do {
       while(current != null)
      {
           push(s, current)
           current = current->left
       }
       poppedNode = pop(s)
       print poppedNode ->info
       current = poppedNode ->right
} while(!isEmpty(s) or current != null)
```



current = **N**

poppedNode =



Inorder Traversal:

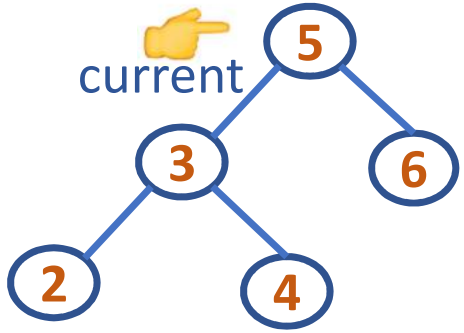
2 3 4 5 6

Iterative Preorder Traversal

```
iterativePreorder(root)
current=root
if (current == null)
   return
s = emptyStack
push(s, current)
while(!isEmpty(s)) {
      current = pop(s)
      print current->info
      //right child is pushed first so that left is processed first
      if(current->right !=NULL)
            push(s, current->right)
      if(current->left !=NULL)
            push(s, current->left)
}
```

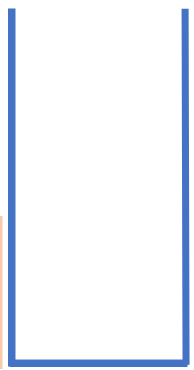
Iterative Preorder Traversal

```
iterativePreorder(root)
current=root
if (current == null)
   return
s = emptyStack
push(s, current)
     ⋮
```



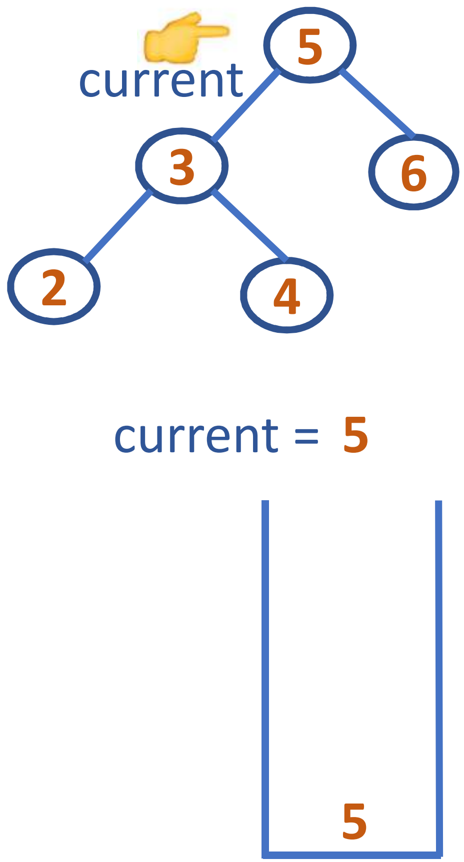
current = **5**

Note: Stack has Address of Nodes Pushed In



Iterative Preorder Traversal

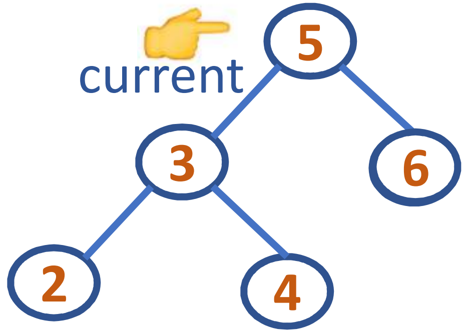
```
iterativePreorder(root)
      ⋮
while (!isEmpty(s))
{
   current = pop(s)
   print current ->info
   if(current->right != null)
      push(s, current->right)
   if(current->left != null)
      push(s, current->left)
}
```



Preorder Traversal:
5

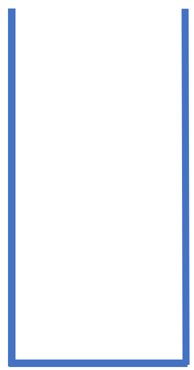
Iterative Preorder Traversal

```
iterativePreorder(root)
      :::
      :::
      :::
while (!isEmpty(s))
{
   current = pop(s)
   print current ->info
   if(current->right != null)
      push(s, current->right)
   if(current->left != null)
      push(s, current->left)
}
```



current = 5

current->right = 6

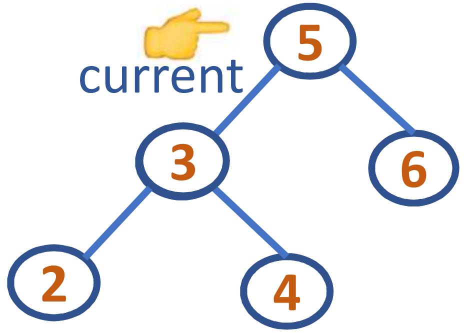


Preorder Traversal:

5

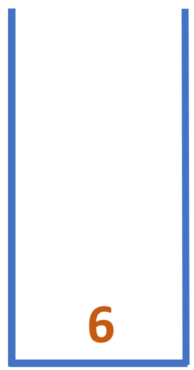
Iterative Preorder Traversal

```
iterativePreorder(root)
    :::
while (!isEmpty(s))
{
   current = pop(s)
   print current ->info
   if(current->right != null)
      push(s, current->right)
   if(current->left != null)
      push(s, current->left)
}
```



current = 5

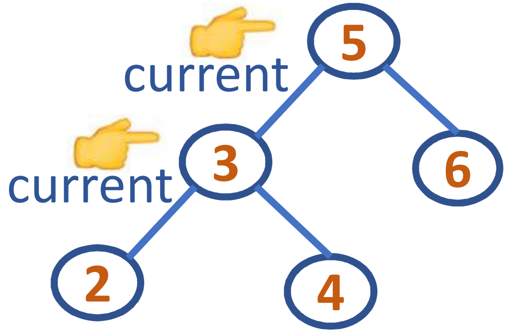
current->left = 3



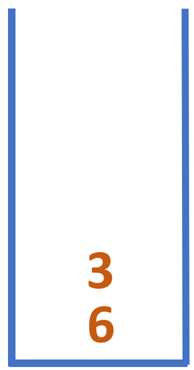
Preorder Traversal:
5

Iterative Preorder Traversal

```
iterativePreorder(root)
    :::
while (!isEmpty(s))
{
   current = pop(s)
   print current ->info
   if(current->right != null)
      push(s, current->right)
   if(current->left != null)
      push(s, current->left)
}
```



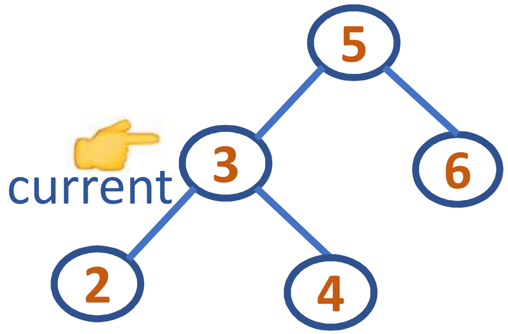
current = 3



Preorder Traversal:
5 3

Iterative Preorder Traversal

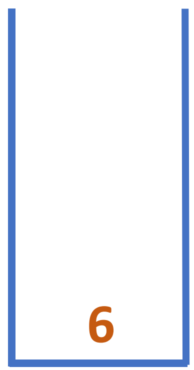
```
iterativePreorder(root)
     ⋮
while (!isEmpty(s))
{
   current = pop(s)
   print current ->info
   if(current->right != null)
      push(s, current->right)
   if(current->left != null)
      push(s, current->left)
}
```



Preorder Traversal:
5 3

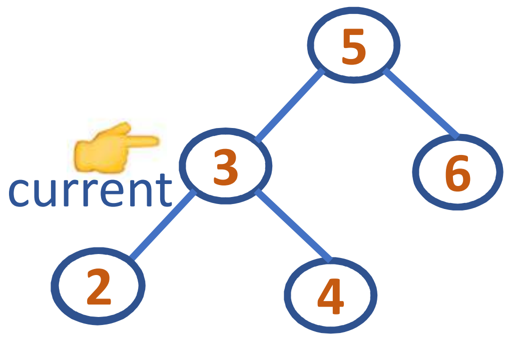
current = **3**

current->right = **4**



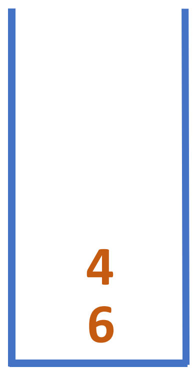
Iterative Preorder Traversal

```
iterativePreorder(root)
    ⋮
while (!isEmpty(s))
{
   current = pop(s)
   print current ->info
   if(current->right != null)
      push(s, current->right)
   if(current->left != null)
      push(s, current->left)
}
```



current = **3**

current->left = **2**

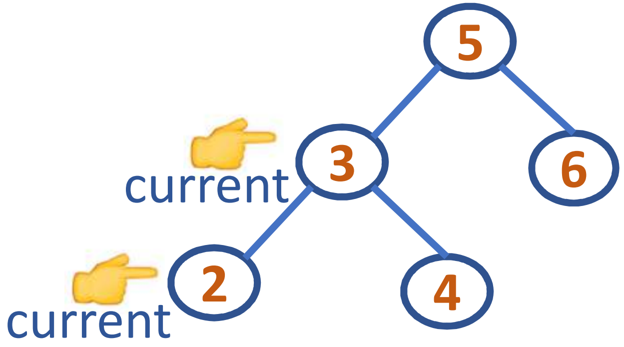


Preorder Traversal:

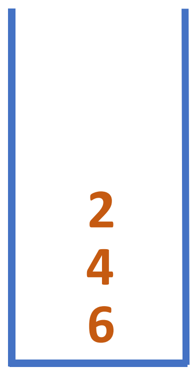
5 3

Iterative Preorder Traversal

```
iterativePreorder(root)
    :::
while (!isEmpty(s))
{
   current = pop(s)
   print current ->info
   if(current->right != null)
      push(s, current->right)
   if(current->left != null)
      push(s, current->left)
}
```



current = 3

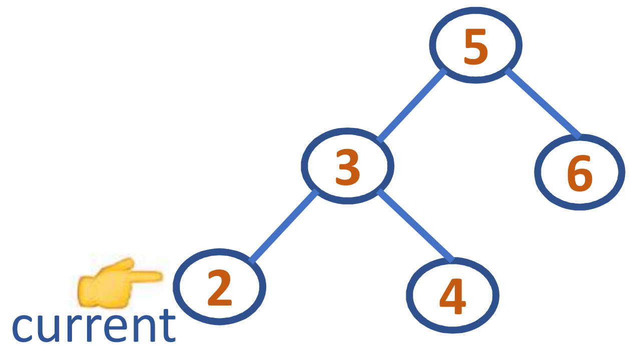


Preorder Traversal:

5 3 2

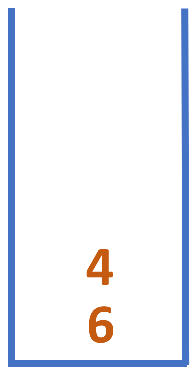
Iterative Preorder Traversal

```
iterativePreorder(root)
      ⋮
while (!isEmpty(s))
{
   current = pop(s)
   print current ->info
   if(current->right != null)
      push(s, current->right)
   if(current->left != null)
      push(s, current->left)
}
```



current = **2**

current->right = **N**

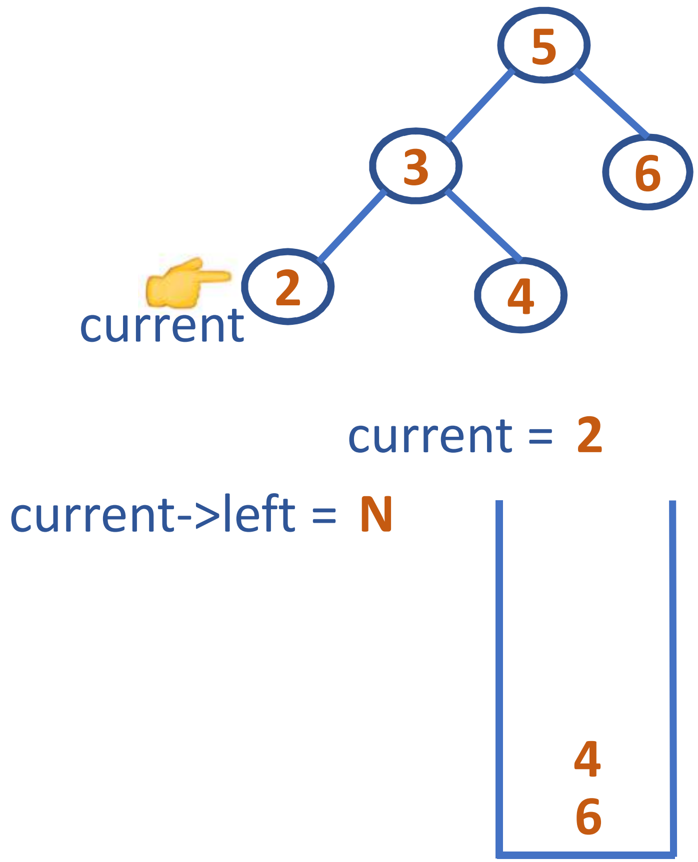


Preorder Traversal:

5 3 2

Iterative Preorder Traversal

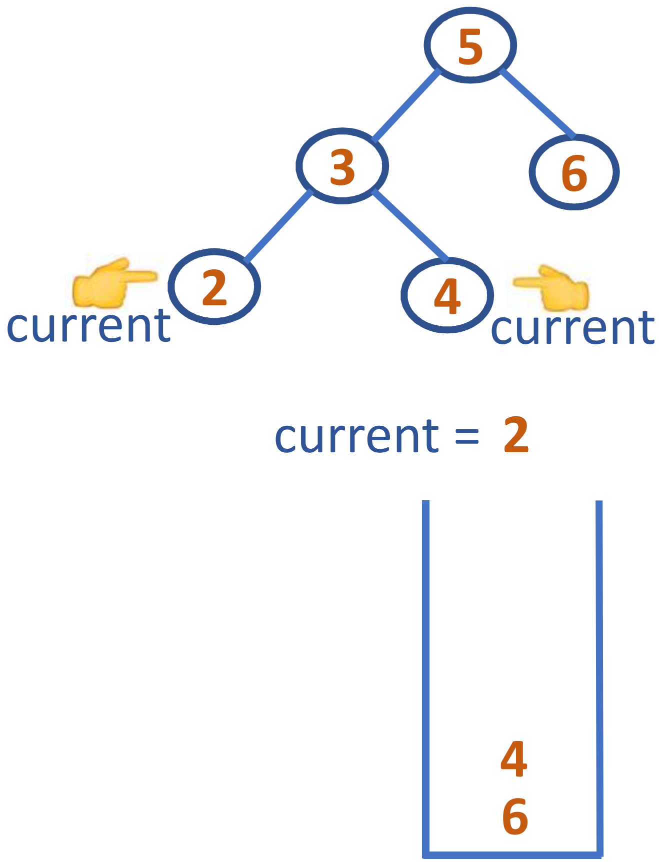
```
iterativePreorder(root)
     :::
while (!isEmpty(s))
{
   current = pop(s)
   print current ->info
   if(current->right != null)
      push(s, current->right)
   if(current->left != null)
      push(s, current->left)
}
```



Preorder Traversal:
5 3 2

Iterative Preorder Traversal

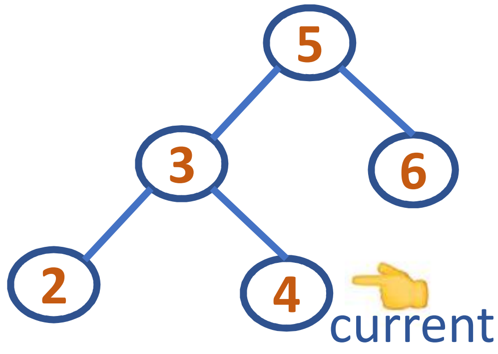
```
iterativePreorder(root)
    :::
while (!isEmpty(s))
{
  current = pop(s)
  print current ->info
  if(current->right != null)
     push(s, current->right)
  if(current->left != null)
     push(s, current->left)
}
```



Preorder Traversal:
5 3 2 4

Iterative Preorder Traversal

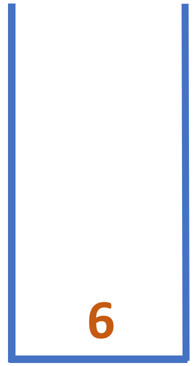
```
iterativePreorder(root)
      :::
while (!isEmpty(s))
{
   current = pop(s)
   print current ->info
   if(current->right != null)
      push(s, current->right)
   if(current->left != null)
      push(s, current->left)
}
```



Preorder Traversal:
5 3 2 4

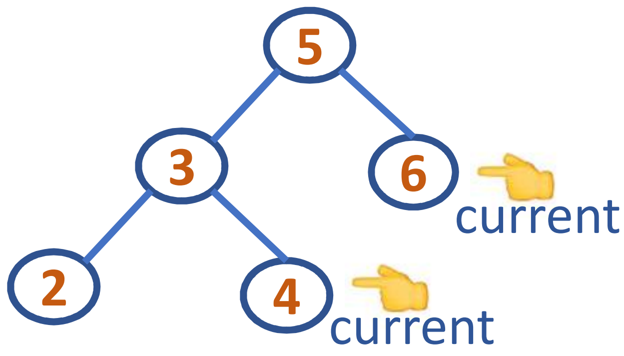
current = **4**

current->right = **N**
current->left = **N**



Iterative Preorder Traversal

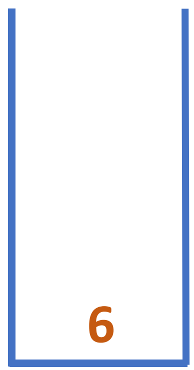
```
iterativePreorder(root)
    :::
while (!isEmpty(s))
{
   current = pop(s)
   print current ->info
   if(current->right != null)
      push(s, current->right)
   if(current->left != null)
      push(s, current->left)
}
```



Preorder Traversal:
5 3 2 4 6

current = **4**

current->right = **N**
current->left = **N**



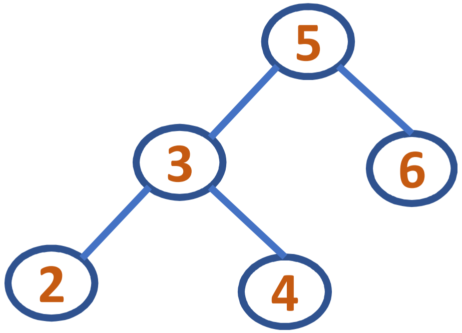
Iterative Postorder Traversal

```
iterativePostorder(root)
s1 = emptyStack ; s2 = emptyStack ; push(s1, root)
while(!isEmpty(s1)) {
    current = pop(s1)
    push(s2,current)
    if(current->left !=NULL)
        push(s1, current->left)
    if(current->right !=NULL)
        push(s1, current->right)
}
while(!isEmpty(s2)) {   //Print all the elements of stack2
    current = pop(s2)
    print current->info
}
```

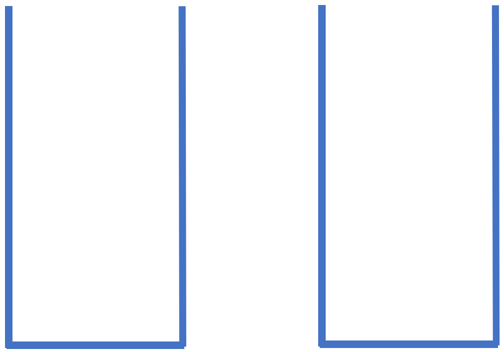
DATA STRUCTURES AND ITS APPLICATIONS

Iterative Postorder Traversal

iterativePostorder(root)
s1 = emptyStack
s2 = emptyStack
push(s1, root)
⋮



root = 5

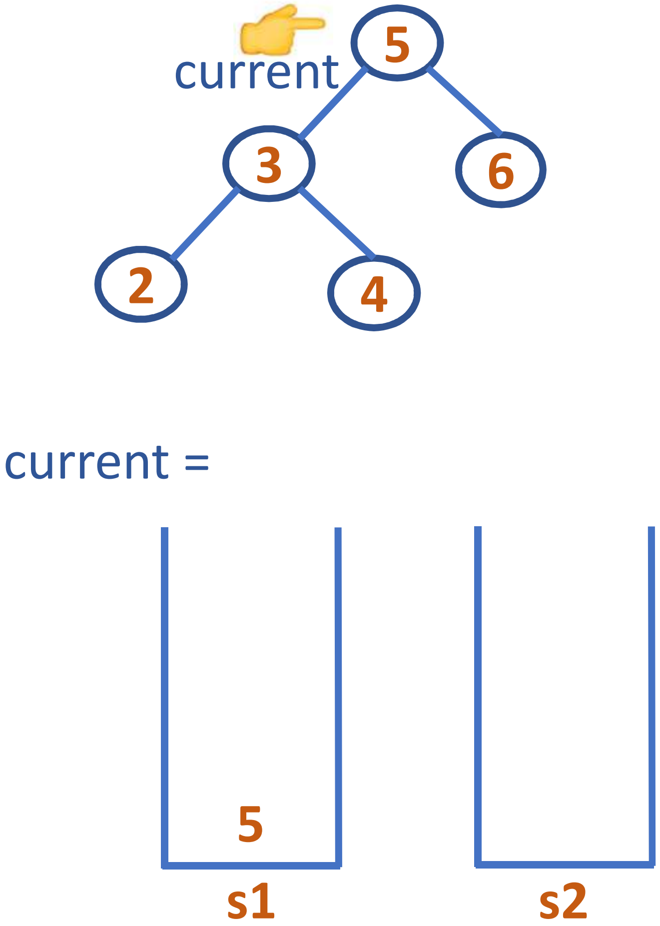


s1 s2

Note: Stacks have Address of Nodes Pushed In

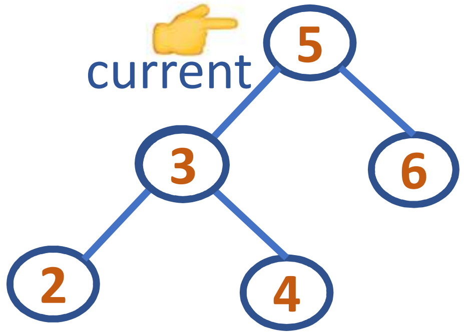
Iterative Postorder Traversal

```
iterativePostorder(root)
    ⁞⁞⁞
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
    ⁞⁞⁞
```



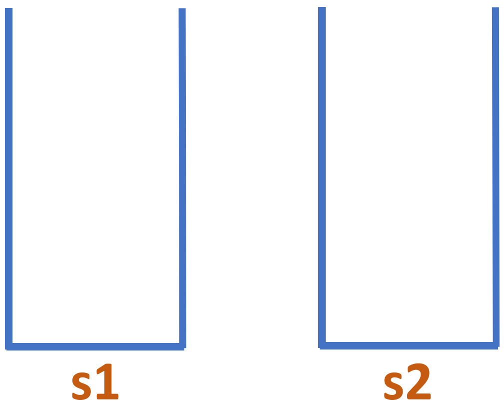
Iterative Postorder Traversal

```
iterativePostorder(root)
    ...
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
    ...
```



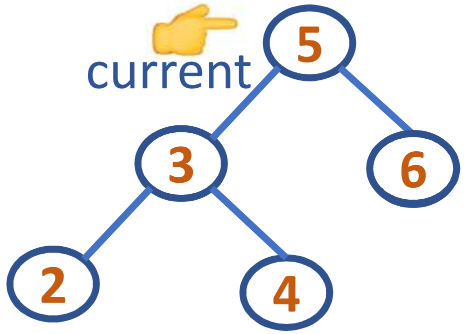
current = **5**

current->left = **3**



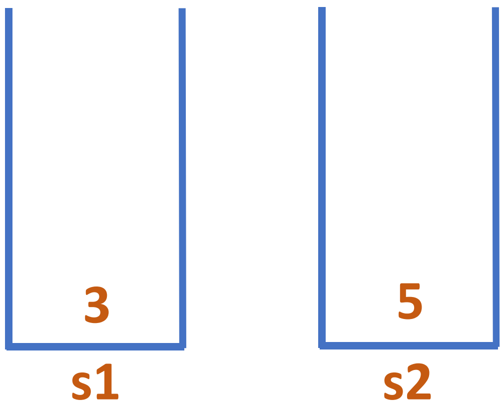
Iterative Postorder Traversal

```
iterativePostorder(root)
    ⋮
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
    ⋮
```



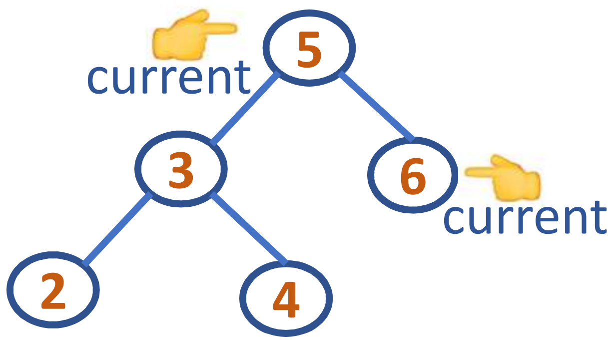
current = **5**

current->right = **6**

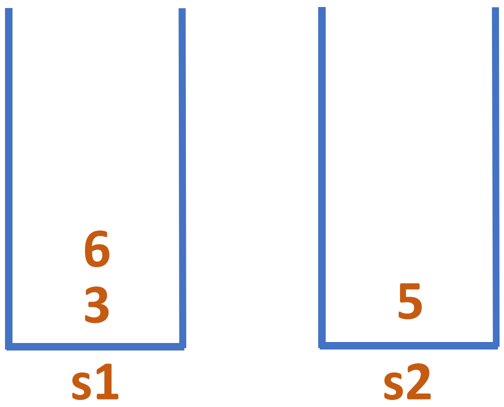


Iterative Postorder Traversal

```
iterativePostorder(root)
    :::
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
    :::
```

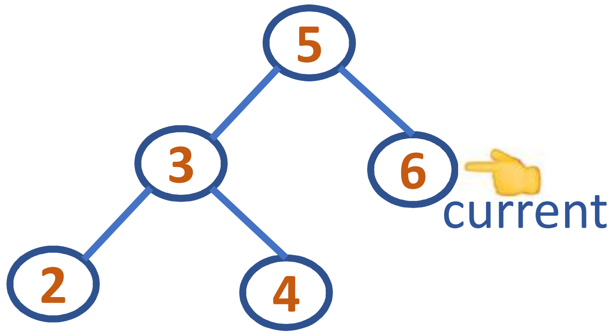


current = 5



Iterative Postorder Traversal

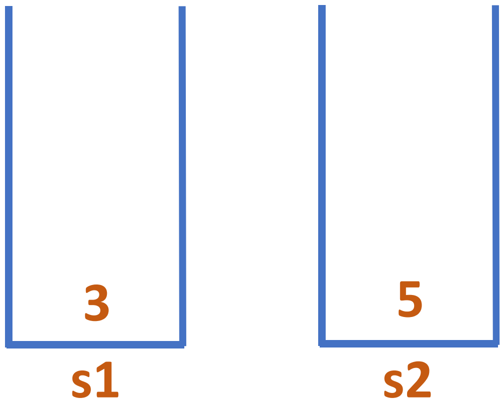
```
iterativePostorder(root)
    ⁝⁝⁝
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
    ⁝⁝⁝
```



current = **6**

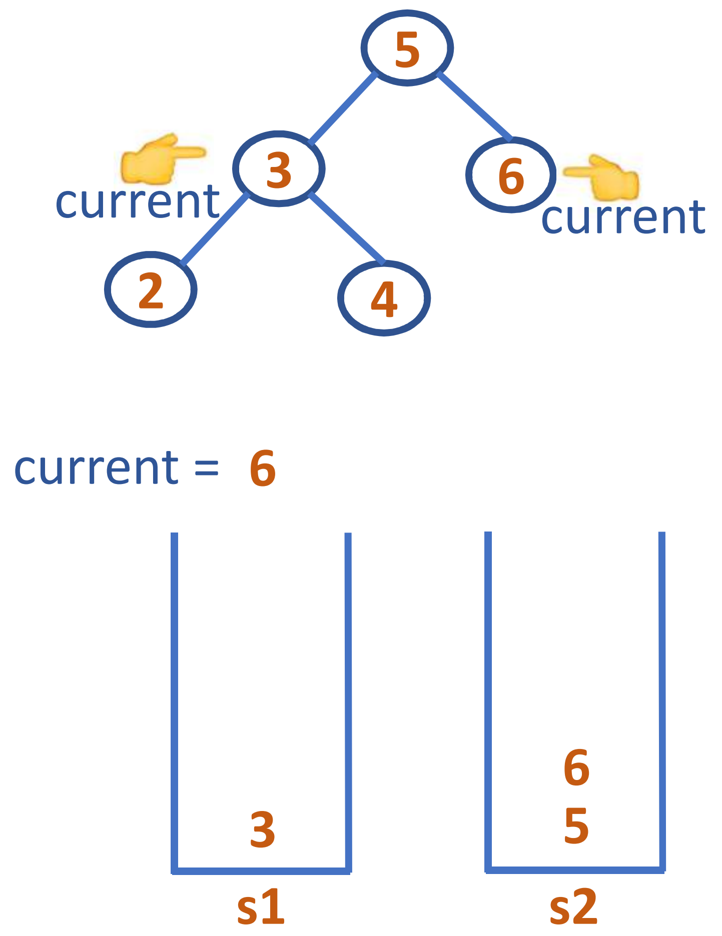
current->left = **N**

current->right = **N**



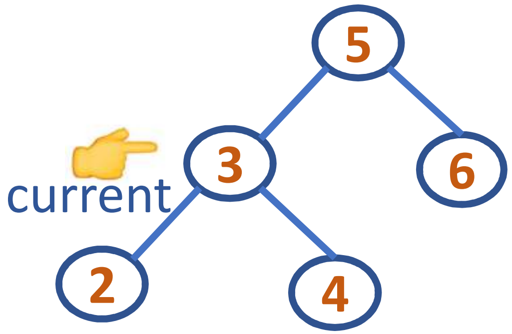
Iterative Postorder Traversal

```
iterativePostorder(root)
    ⋮
while(!isEmpty(s1))
{
  current = pop(s1)
  push(s2,current)
  if(current->left !=NULL)
    push(s1, current->left)
  if(current->right !=NULL)
    push(s1, current->right)
}
    ⋮
```

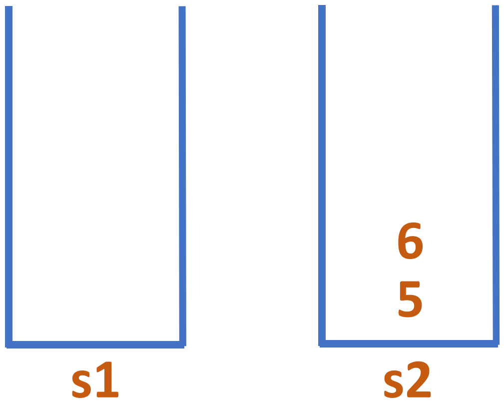


Iterative Postorder Traversal

```
iterativePostorder(root)
    :::
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
    :::
```

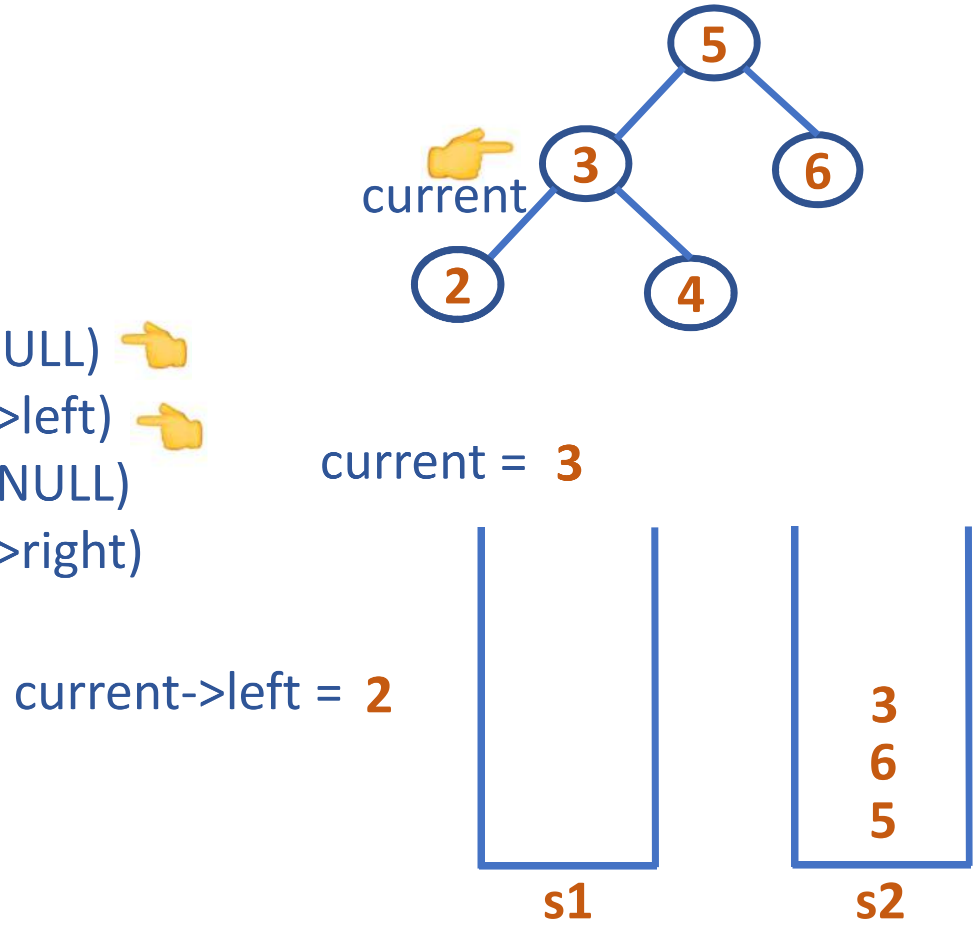


current = **3**



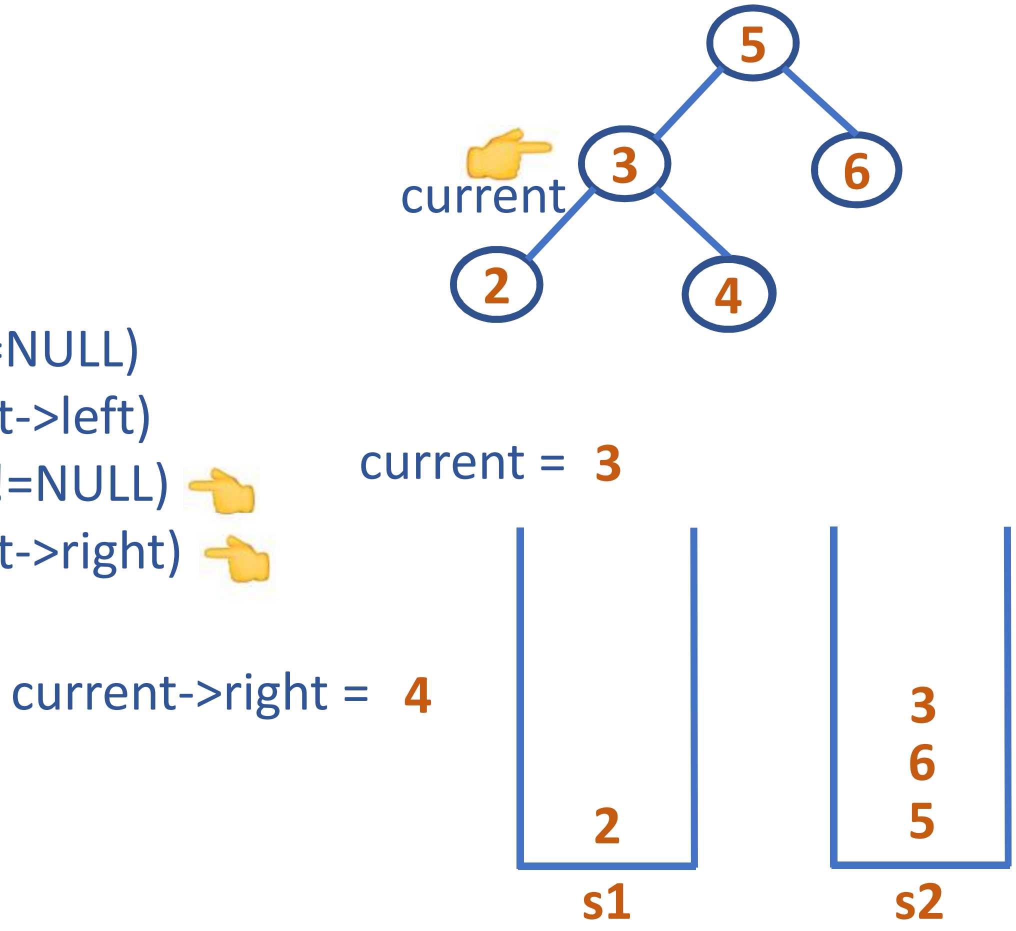
Iterative Postorder Traversal

```
iterativePostorder(root)
    ⁞⁞⁞
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
    ⁞⁞⁞
```



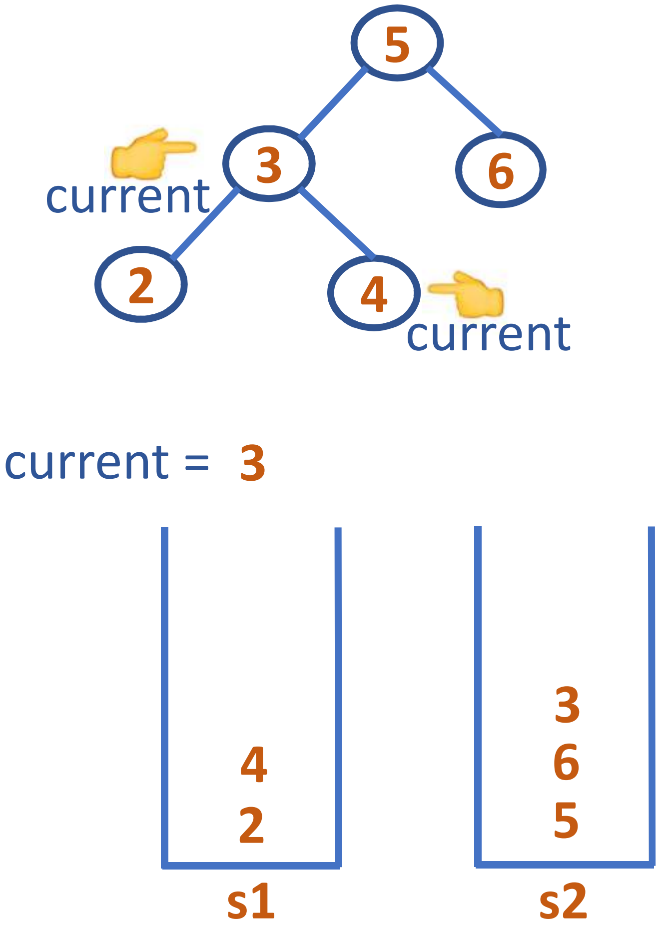
Iterative Postorder Traversal

```
iterativePostorder(root)
    ⁝⁝⁝
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
    ⁝⁝⁝
```



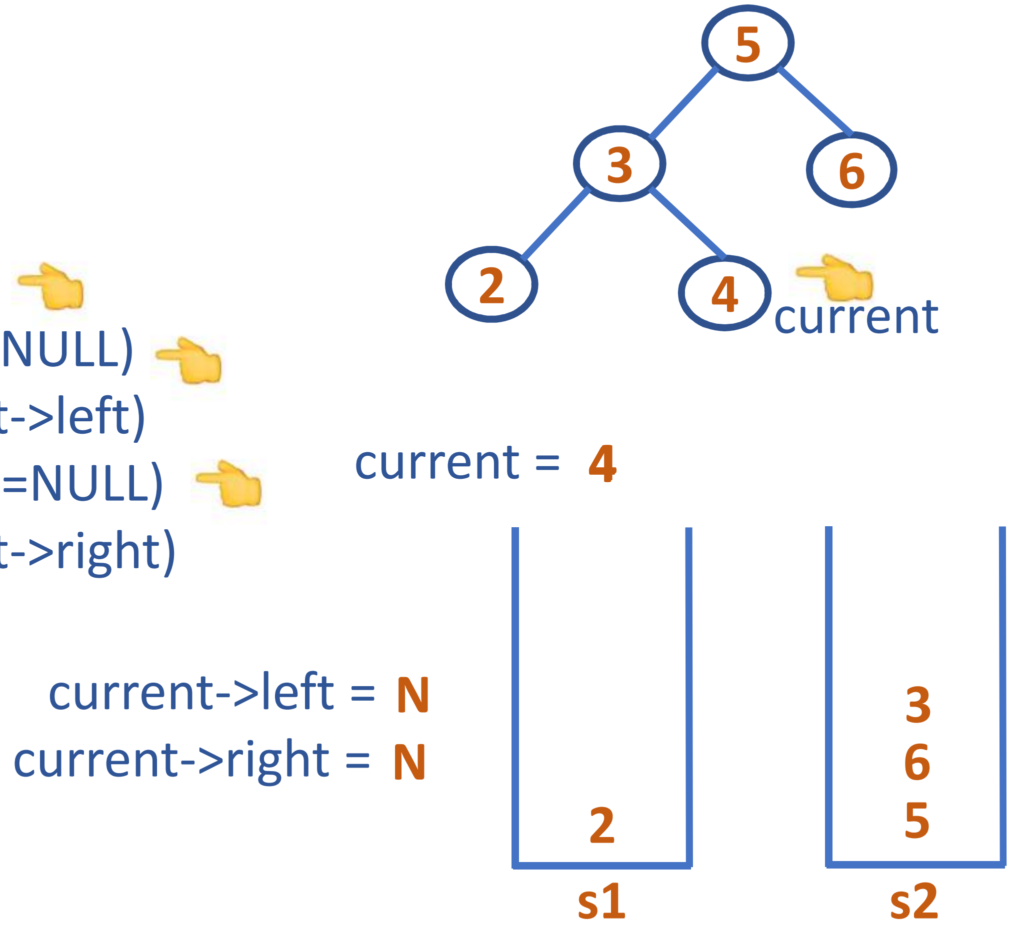
Iterative Postorder Traversal

```
iterativePostorder(root)
    ⋮
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
    ⋮
```



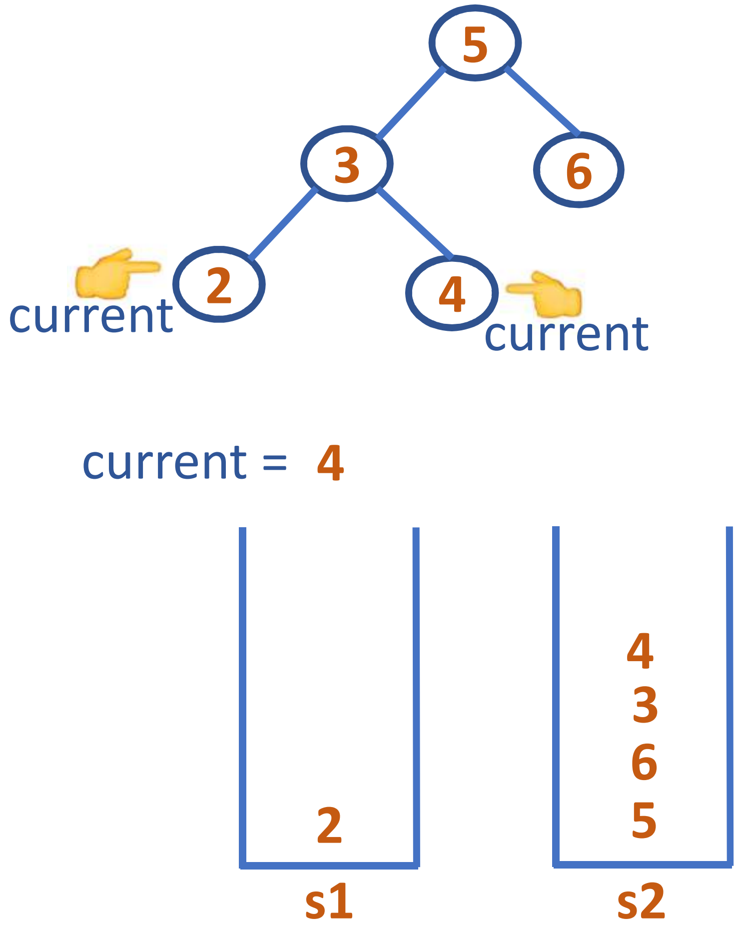
Iterative Postorder Traversal

```
iterativePostorder(root)
    ⁞⁞⁞
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
    ⁞⁞⁞
```



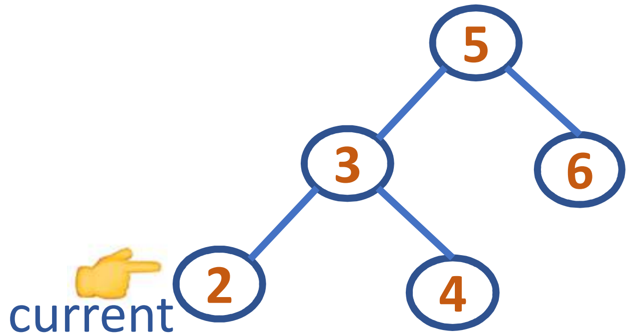
Iterative Postorder Traversal

```
iterativePostorder(root)
    ⁝⁝⁝
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
    ⁝⁝⁝
```



Iterative Postorder Traversal

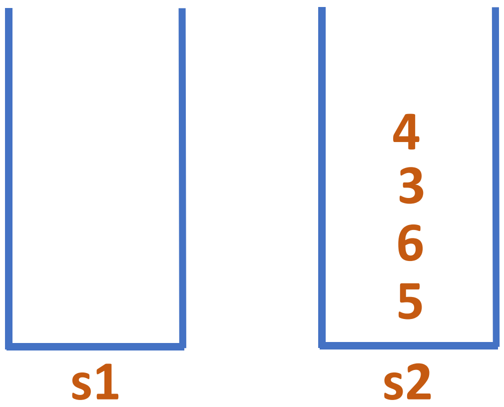
```
iterativePostorder(root)
    ::::
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
    ::::
```



current = **2**

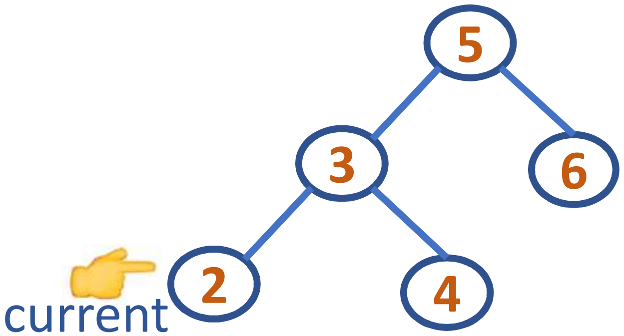
current->left = **N**

current->right = **N**

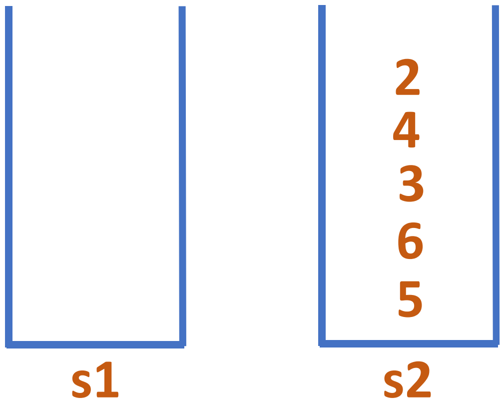


Iterative Postorder Traversal

```
iterativePostorder(root)
    ::::
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
while(!isEmpty(s2)) {
     current = pop(s2)
     print current->info
}
```

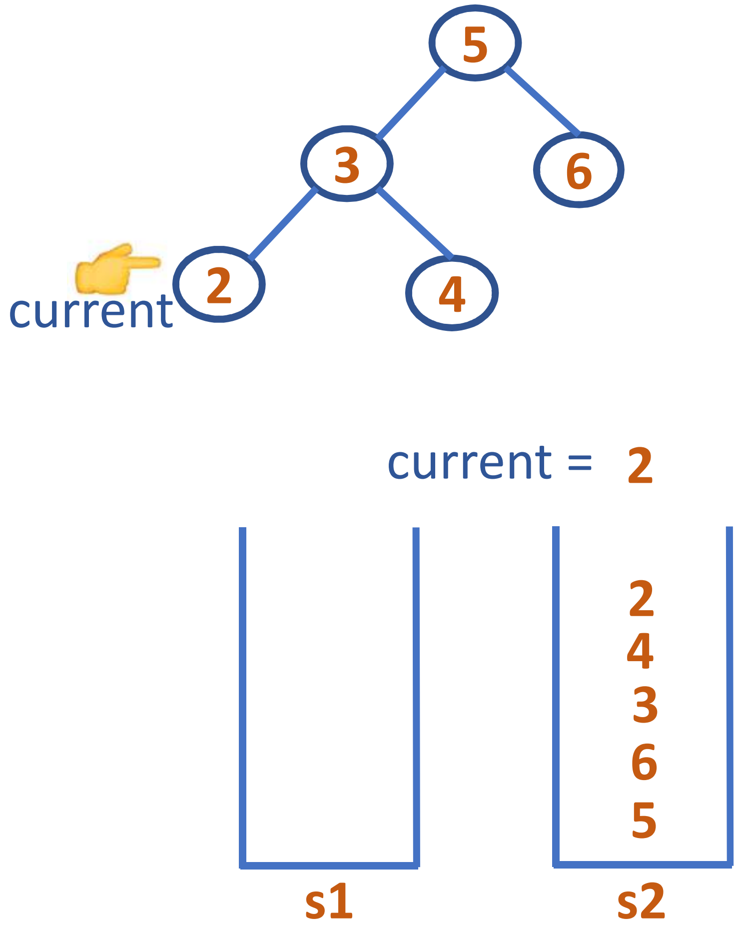


current = **2**



Iterative Postorder Traversal

```
iterativePostorder(root)
      ⋮
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
while(!isEmpty(s2)) {
     current = pop(s2)
     print current->info
}
```

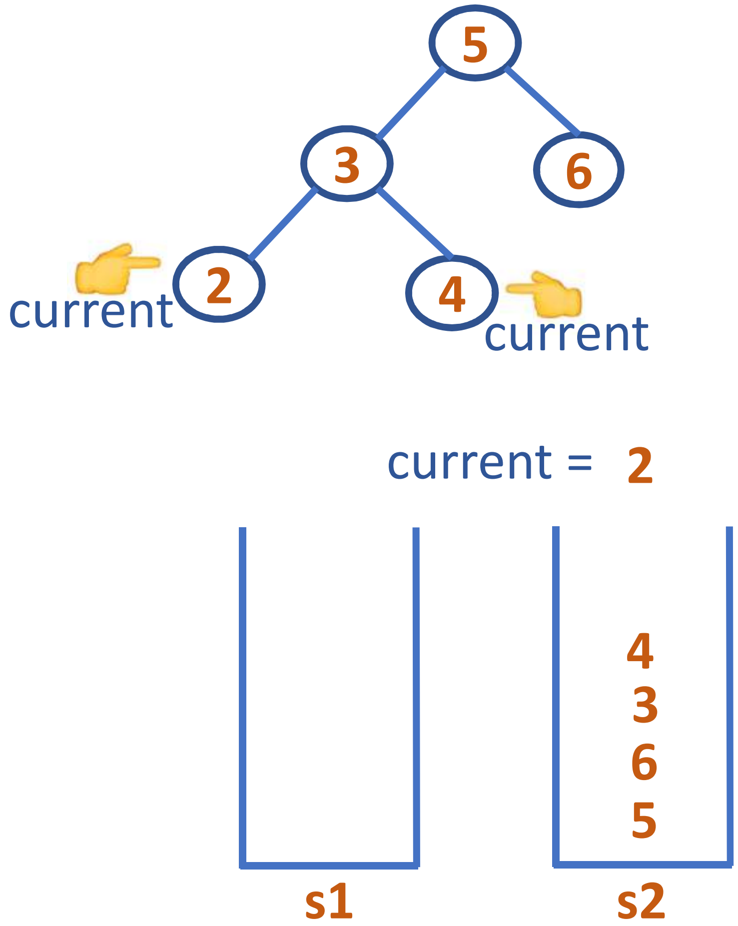


Postorder Traversal:

2

Iterative Postorder Traversal

```
iterativePostorder(root)
    ⋮
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
while(!isEmpty(s2)) {
     current = pop(s2)
     print current->info
}
```

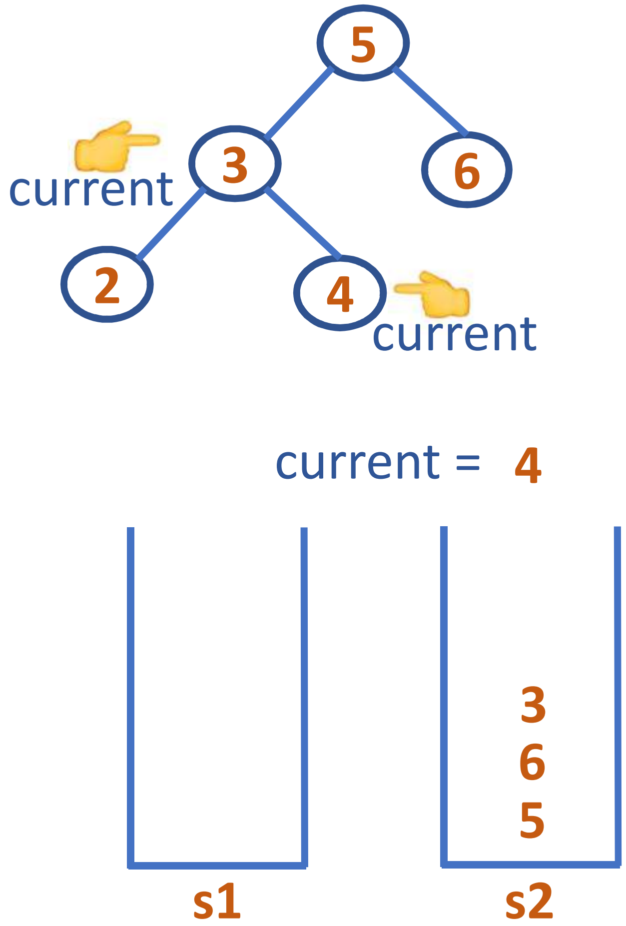


Postorder Traversal:

2 4

Iterative Postorder Traversal

```
iterativePostorder(root)
     ⁞
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
while(!isEmpty(s2)) {
     current = pop(s2)
     print current->info
}
```

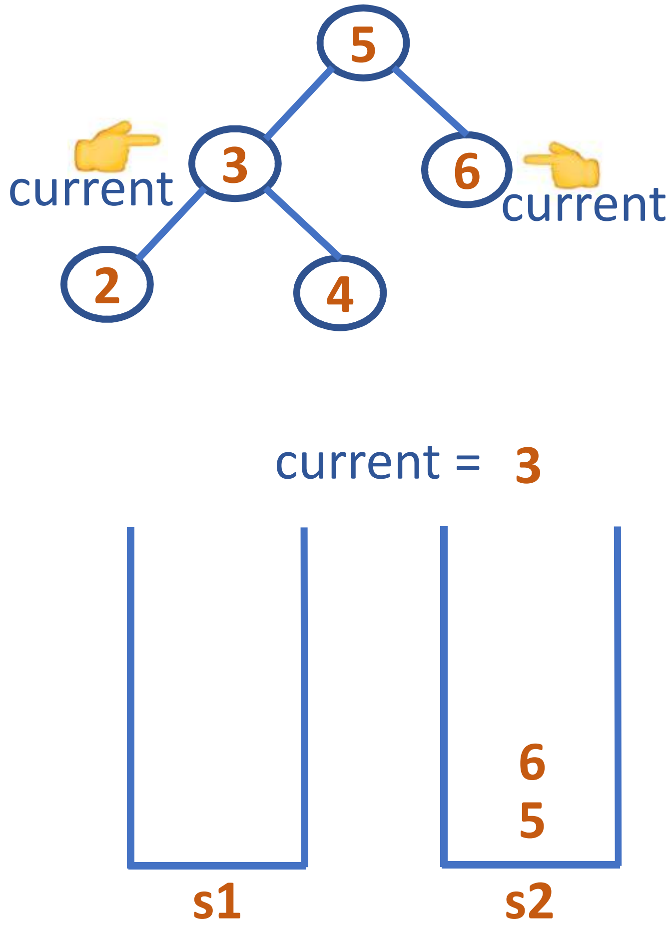


Postorder Traversal:

2 4 3

Iterative Postorder Traversal

```
iterativePostorder(root)
     ⋮⋮
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
while(!isEmpty(s2)) {
     current = pop(s2)
     print current->info
}
```

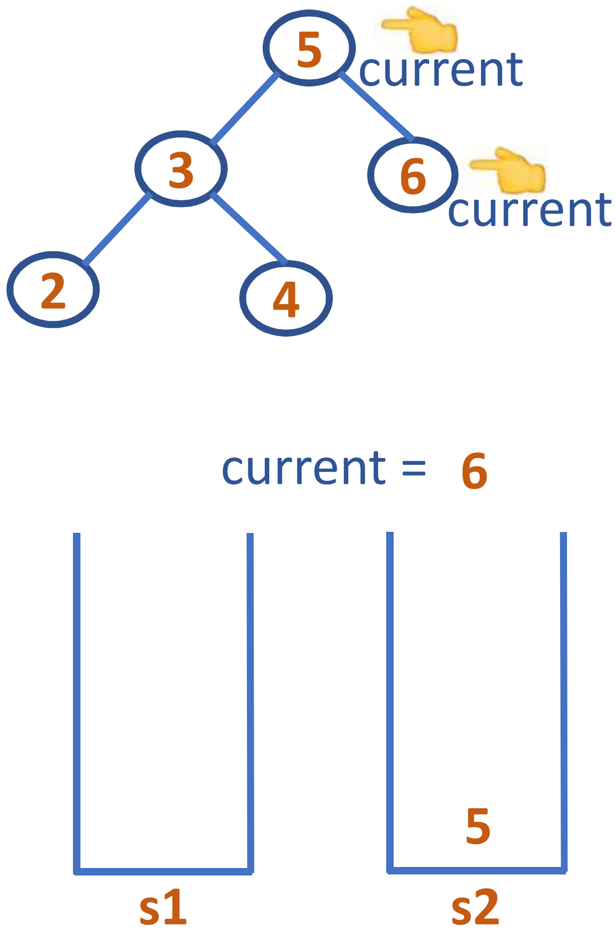


Postorder Traversal:

2 4 3 6

Iterative Postorder Traversal

```
iterativePostorder(root)
     ::::
while(!isEmpty(s1))
{
   current = pop(s1)
   push(s2,current)
   if(current->left !=NULL)
     push(s1, current->left)
   if(current->right !=NULL)
     push(s1, current->right)
}
while(!isEmpty(s2)) {
     current = pop(s2)
     print current->info
}
```



Postorder Traversal:

2 4 3 6 5

THANK YOU

Shylaja S S

Department of Computer Science
& Engineering

shylaja.sharath@pes.edu

+91 9449867804

DATA STRUCTURES AND ITS APPLICATIONS

UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Threaded BST and its Implementation

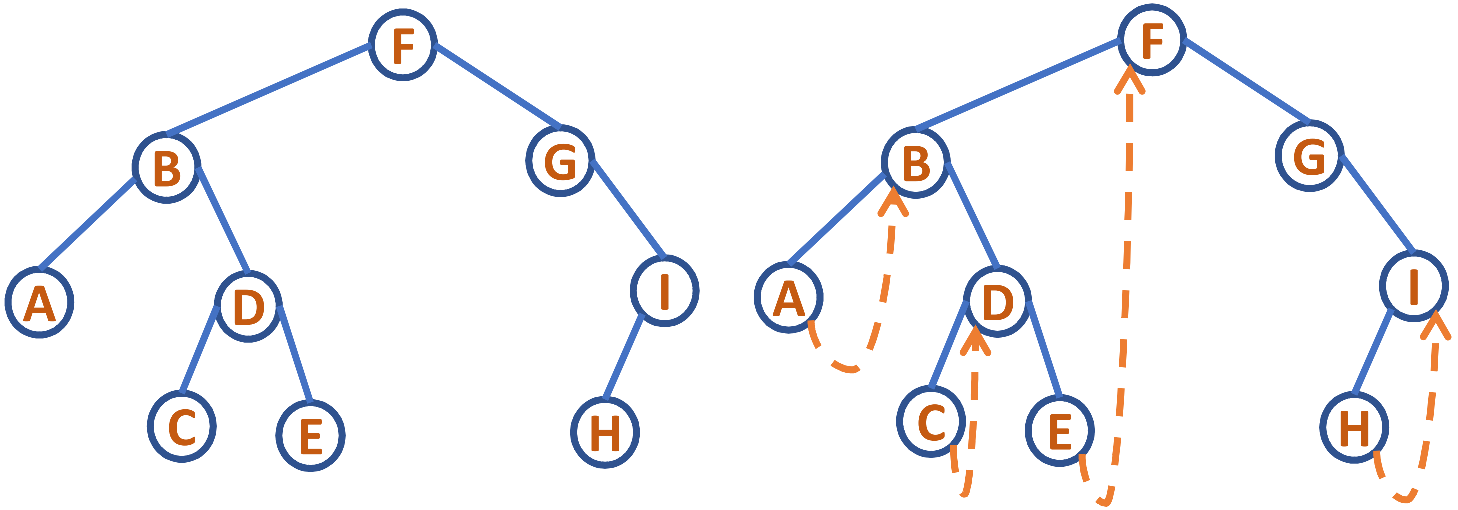
Shylaja S S
Department of Computer Science & Engineering

Motivation

- Iterative Inorder Traversal requires Explicit stack
- Costly
- Since we loose track of address as and when we navigate, Node addresses were stacked
- If this can be achieved through some other less expensive mechanism, we can eliminate the use of explicit stack
- Small structural modification carried on Binary tree will solve the above problem

- We can use the right pointer of a node to point to the inorder successor if in case it is not pointing to the child. Such a tree is called **Right-In Threaded** Binary Tree
- If we use the left pointer to store the inorder predecessor, the tree is called **Left-In Threaded** Binary Tree
- If we use both the pointers, the tree is called **In Threaded** Binary Tree

Right-In Threaded Binary Tree

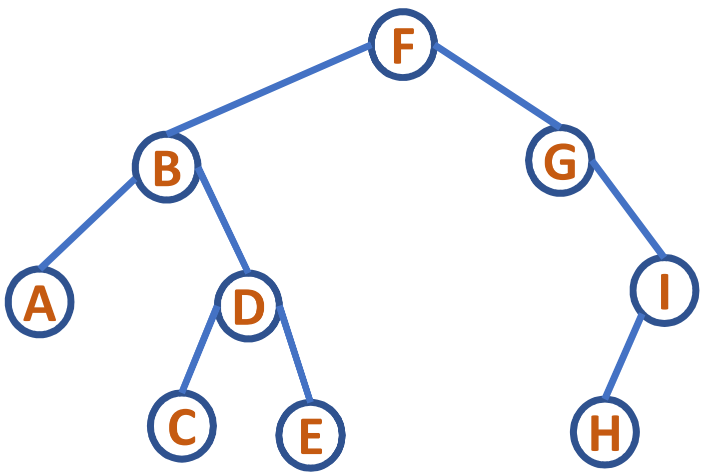


Binary Tree Right-In Threaded Binary Tree

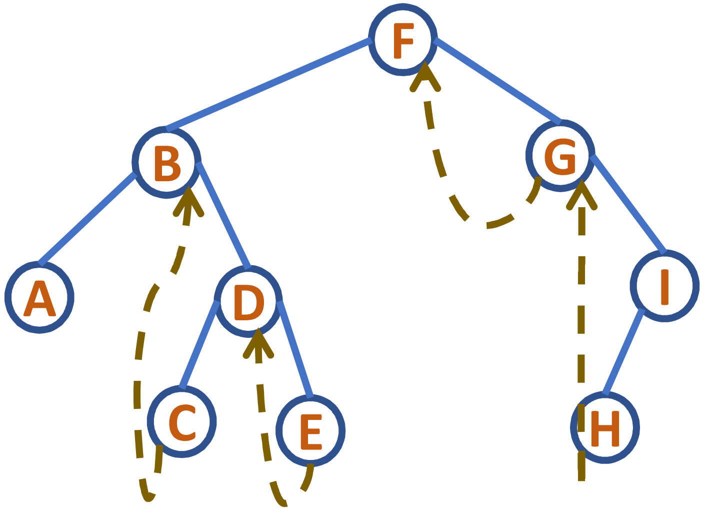
Inorder Traversal:
A B C D E F G H I

Nodes with Right Pointer NULL	A	C	E	H	I
Inorder Successor	B	D	F	I	-

Left-In Threaded Binary Tree



Binary Tree

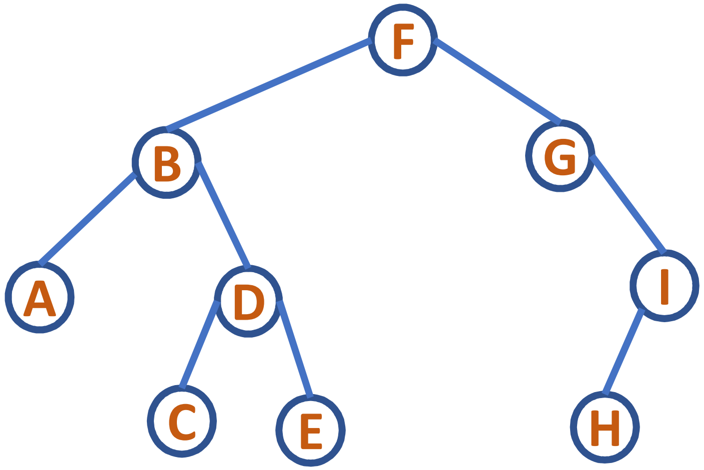


Left-In Threaded Binary Tree

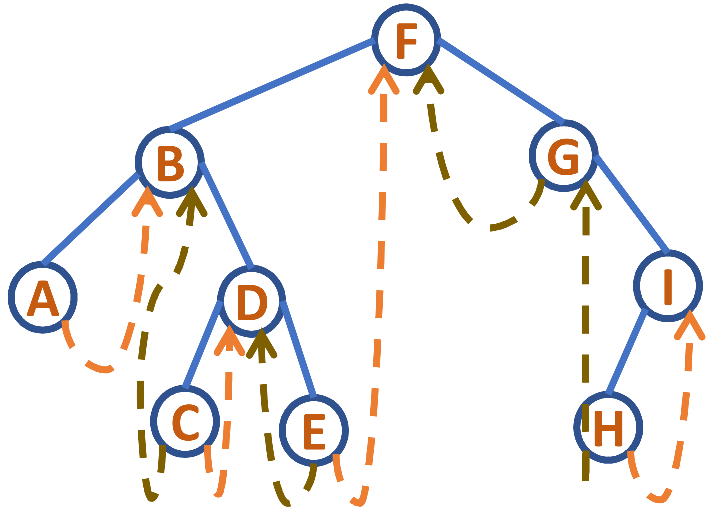
Inorder Traversal:
A B C D E F G H I

Nodes with Left Pointer NULL	A	C	E	G	H
Inorder Predecessor	-	B	D	F	G

In Threaded Binary Tree



Binary Tree



In Threaded Binary Tree

Right In Threaded Binary Tree

```
typedef struct node
{
        int info;
        struct node *left;      // pointer to left child
        struct node *right;     // pointer to right child
        int rthread;            // rthread is TRUE if right is NULL
                                // or a non-NULL thread
}NODE;
```

Node Structure

info	left	right	rthread

Threaded Binary Search Tree: Implementation

```
NODE* createNode(int e)
{
        NODE* temp=malloc(sizeof(NODE));
        temp->info=e;
        temp->left=NULL;
        temp->right=NULL;
        temp->rthread=1;
        return temp;          // Returns: 2000
}
```

info	left	right	rthread

createNode(57)

temp ->

57	NULL	NULL	1

Let Address of this node on Heap: 2000

Threaded Binary Search Tree: Implementation

Right In Threaded Binary Tree: 57, 25, 28

- A node is created with rthread set to TRUE
- **insert 57**

Address: 800

57

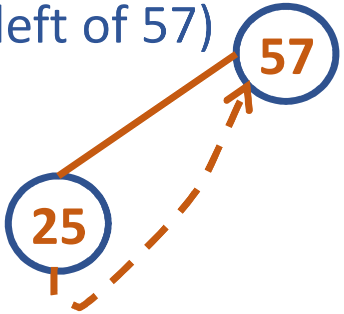
Node Structure

info	left	right	rthread
57	NULL	NULL	1

Threaded Binary Search Tree: Implementation

Right In Threaded Binary Tree: 57, 25, 28

- A node is created with rthread set to TRUE
- insert 57
- **insert 25** (left of 57)



Node Structure

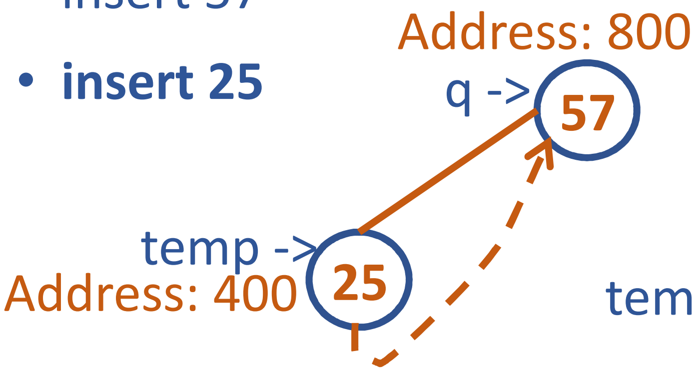
info	left	right	rthread
57	NULL 400	NULL	1
25	NULL	NULL 800	1

Threaded Binary Search Tree: Implementation

PES UNIVERSITY ONLINE

Right In Threaded Binary Tree: 57, 25, 28

- A node is created with rthread set to TRUE
- insert 57
- **insert 25**



Node Structure

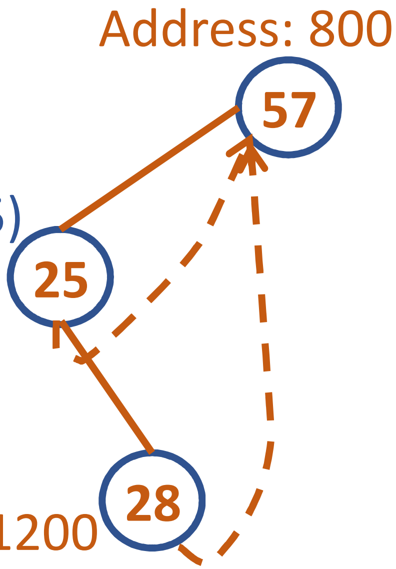
	info	left	right	rthread
q ->	57	NULL 400	NULL	1
temp ->	25	NULL	NULL 800	1

```
void setLeft(NODE* q, int e) {      //set node with info e to left of q
        NODE* temp=createNode(e);           //e=25
        q->left=temp;
        temp->right=q;
}
```

Threaded Binary Search Tree: Implementation

Right In Threaded Binary Tree: 57, 25, 28

- A node is created with rthread set to TRUE
- insert 57
- insert 25
- **insert 28**
  (Right of 25)



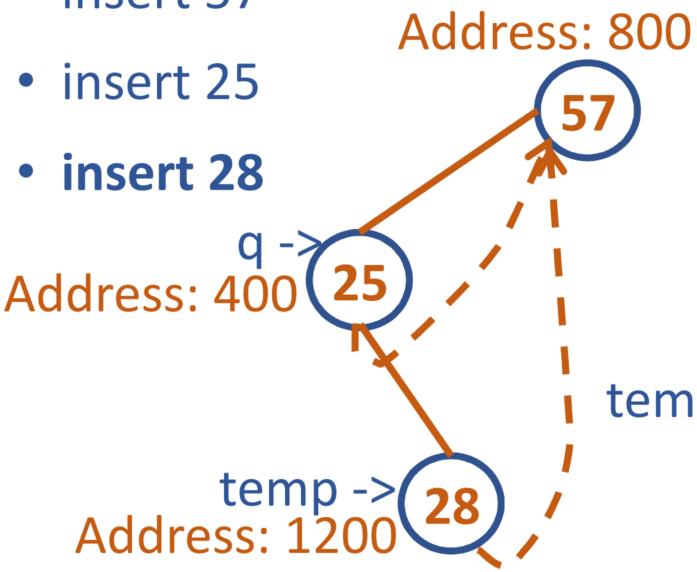
Node Structure

info	left	right	rthread
57	400	NULL	1
25	NULL	1200	0
28	NULL	800	1

Threaded Binary Search Tree: Implementation

Right In Threaded Binary Tree: 57, 25, 28

- A node is created with rthread set to TRUE
- insert 57
- insert 25
- **insert 28**



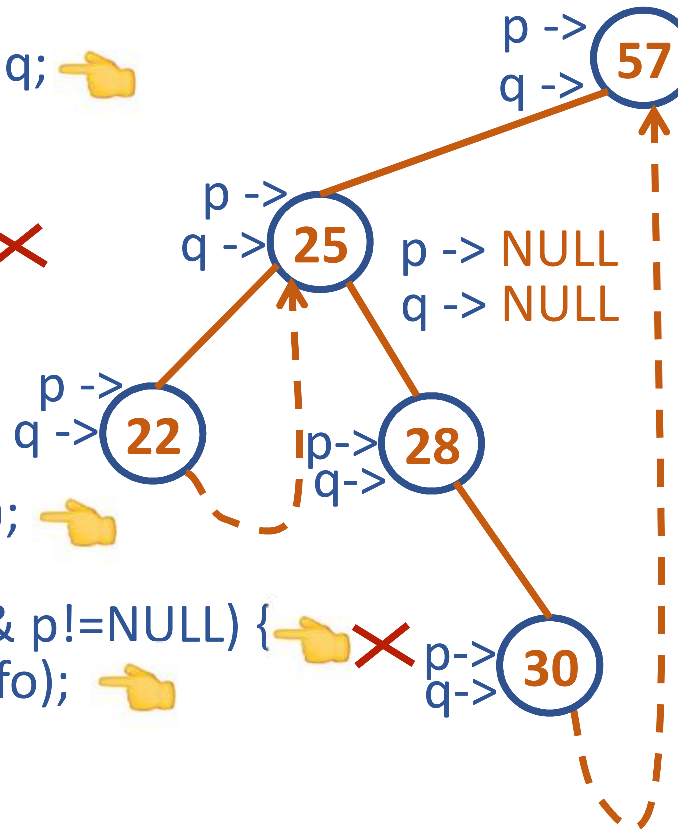
Node Structure

	info	left	right	rthread
	57	400	NULL	1
q ->	25	NULL	1200 800	0 1
temp ->	28	NULL	800 NULL	1

```
void setRight(NODE* q,int e) {
   NODE* temp=createNode(e);
   temp->right=q->right;
   q->right=temp;
   q->rthread=0;
}
```

Threaded Binary Search Tree: Inorder Traversal

```
void inOrder(NODE *root) {
   NODE *p=root;    NODE *q;
   do{
         q=NULL;
         while(p!=NULL) {
            q=p;
            p=p->left;
         }
         if(q!=NULL) {
            printf("%d ",q->info);
            p=q->right;
            while(q->rthread && p!=NULL) {
               printf("%d ",p->info);
               q=p;
               p=p->right;
            }
         }
   }while(q!=NULL);
}
```



q -> NULL

rthread is TRUE for nodes with info: 22, 30, 57

Inorder Traversal:

22 25 28 30 57

THANK YOU

Shylaja S S

Department of Computer Science
& Engineering

shylaja.sharath@pes.edu

+91 9449867804

DATA STRUCTURES AND ITS APPLICATIONS
UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

PES UNIVERSITY ONLINE

DATA STRUCTURES AND ITS APPLICATIONS

Implementation of Binary Expression Tree

Shylaja S S

Department of Computer Science & Engineering

Expression Tree

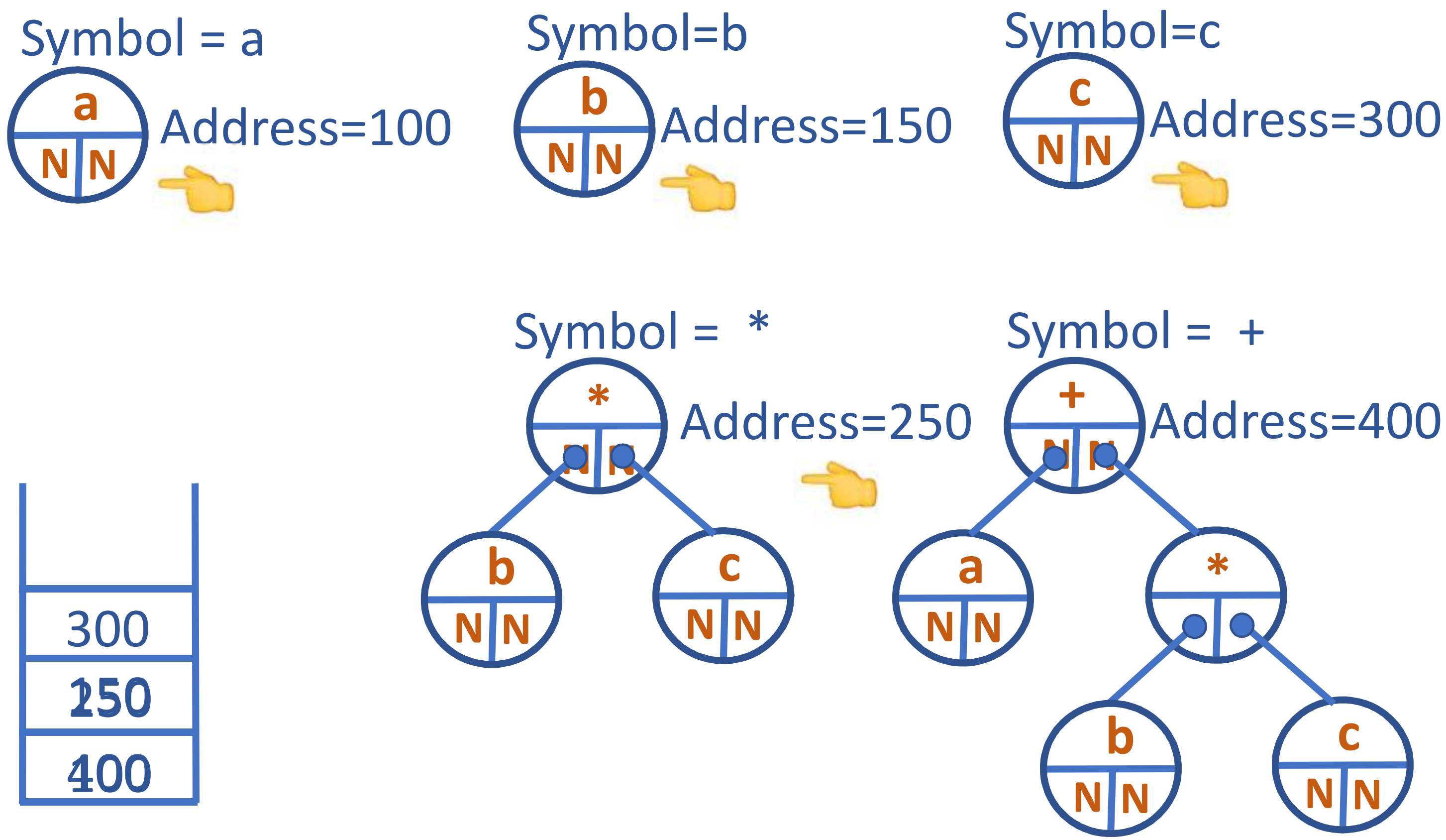
- An expression can be represented using the **Expression Tree** data structure
- Such a tree is built normally for translating the code as data and then analysing and evaluating expressions
- **Immutable**: To change the expression another tree has to be constructed

Expression Tree Construction

- Normally a postfix expression is used in constructing the Expression tree
- When an operand is received, a new node is created which will be a leaf in the expression tree
- If an operator, it connects to two leaves
- Stack DS is used as intermediary storing place of node's address

Expression Tree Construction

Postfix Expression: abc*+



Expression Tree Construction

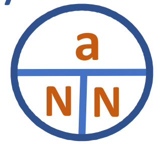
- Scan the postfix expression till the end, one symbol at a time
  - Create a new node, with symbol as info and left and right link as NULL
  - If symbol is an operand, push address of node to stack
  - If symbol is an operator
    - Pop address from stack and make it right child of new node
    - Pop address from stack and make it left child of new node
    - Now push address of new node to stack
- Finally, stack has only element which is the address of the root of expression tree

Expression Tree Construction

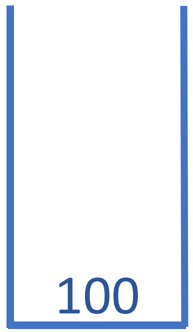
Postfix Expression: **a b c * +**

- Scan the postfix expression till the end, one symbol at a time
  - Create a new node, with symbol as info and left and right link as NULL
  - If symbol is an operand, push address of node to stack
  - If symbol is an operator
    - Pop the address from stack and make it right child of new node
    - Pop the address from stack and make it left child of new node
    - Now push address of new node to stack
- Finally, stack has only element which is the address of the root of expression tree

Symbol = a



Address=100

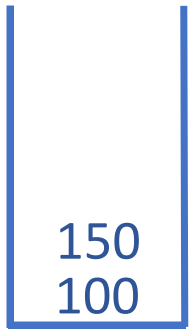


Expression Tree Construction

Postfix Expression: **a b c * +**

- Scan the postfix expression till the end, one symbol at a time
  - Create a new node, with symbol as info and left and right link as NULL
  - If symbol is an operand, push address of node to stack
  - If symbol is an operator
    - Pop the address from stack and make it right child of new node
    - Pop the address from stack and make it left child of new node
    - Now push address of new node to stack
- Finally, stack has only element which is the address of the root of expression tree

Symbol = b

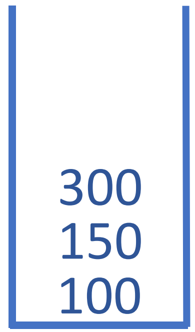


Expression Tree Construction

Postfix Expression: **a b c * +**

- Scan the postfix expression till the end, one symbol at a time
  - Create a new node, with symbol as info and left and right link as NULL
  - If symbol is an operand, push address of node to stack
  - If symbol is an operator
    - Pop the address from stack and make it right child of new node
    - Pop the address from stack and make it left child of new node
    - Now push address of new node to stack
- Finally, stack has only element which is the address of the root of expression tree

Symbol = c

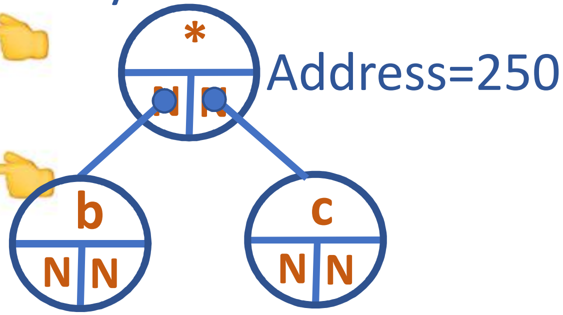


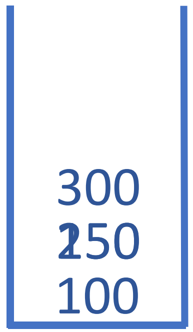
Expression Tree Construction

Postfix Expression: **a b c * +**

- Scan the postfix expression till the end, one symbol at a time
  - Create a new node, with symbol as info and left and right link as NULL
  - If symbol is an operand, push address of node to stack
  - If symbol is an operator
    - Pop the address from stack and make it right child of new node
    - Pop the address from stack and make it left child of new node
    - Now push address of new node to stack
- Finally, stack has only element which is the address of the root of expression tree

Symbol = *

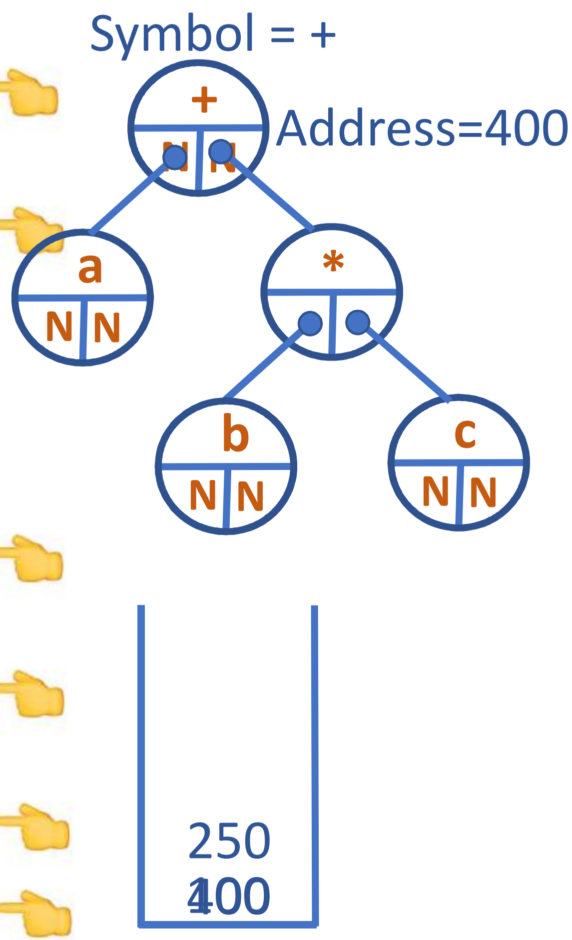




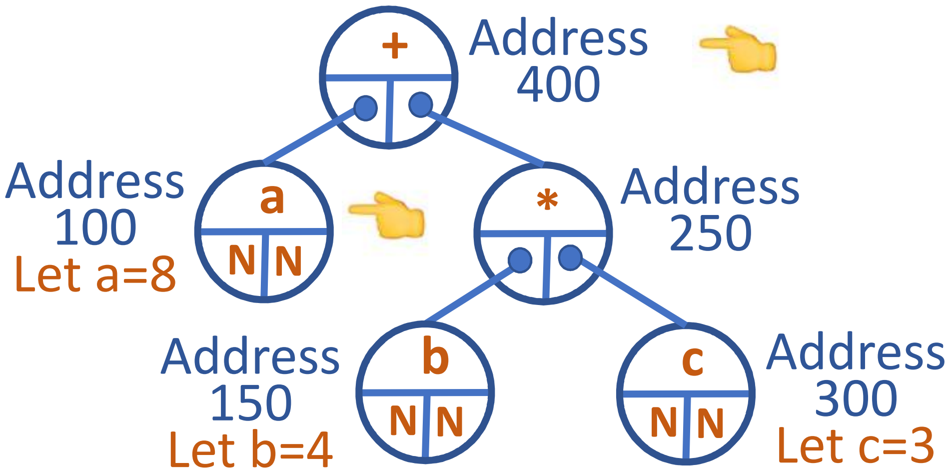
Expression Tree Construction

Postfix Expression: **a b c * +**

- Scan the postfix expression till the end, one symbol at a time
  - Create a new node, with symbol as info and left and right link as NULL
  - If symbol is an operand, push address of node to stack
  - If symbol is an operator
    - Pop the address from stack and make it right child of new node
    - Pop the address from stack and make it left child of new node
    - Now push address of new node to stack
- Finally, stack has only element which is the address of the root of expression tree



Expression Tree Evaluation



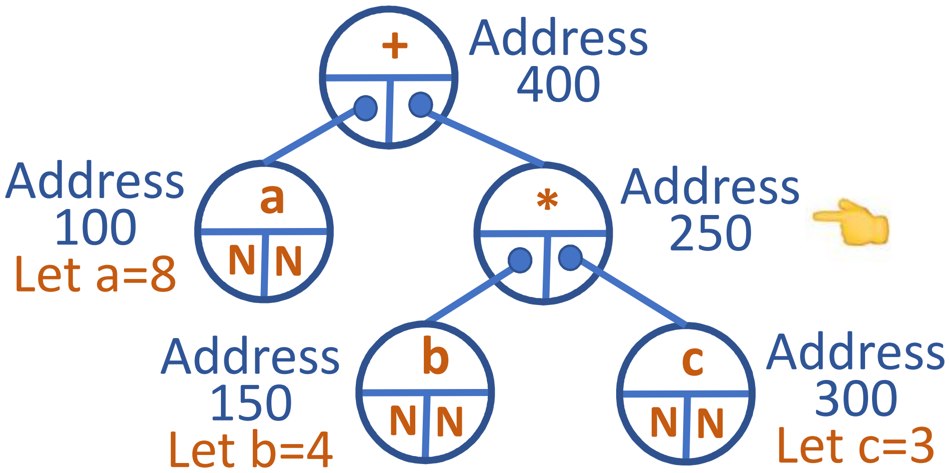
eval(400)
  return eval(100) + eval(250)

eval(100)
  return **8**

- Think in terms of recursion

```
eval(t) // 't' has the address of the root node of expression tree
  if t->data is an operator
    return eval (t->left) t->data eval(t->right)
  return t->data
```

Expression Tree Evaluation



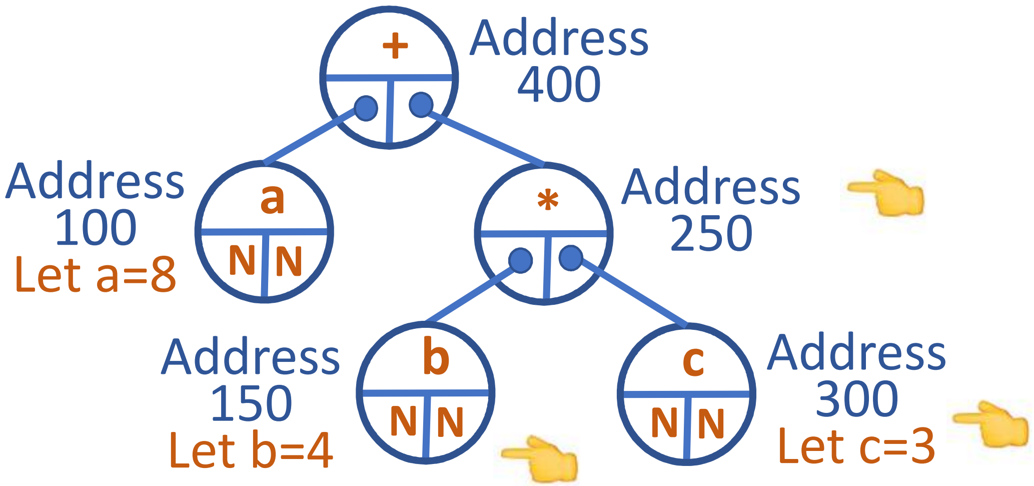
eval(400)
return 8 + eval(250)

eval(250)
return eval(150) * eval(300)

- Think in terms of recursion

```
eval(t)// 't' has the address of the root node of expression tree
  if t->data is an operator
    return eval (t->left) t->data eval(t->right)
  return t->data
```

Expression Tree Evaluation



eval(400)
return 8 + eval(250)

eval(250)
return eval(4150) * eval(3300)

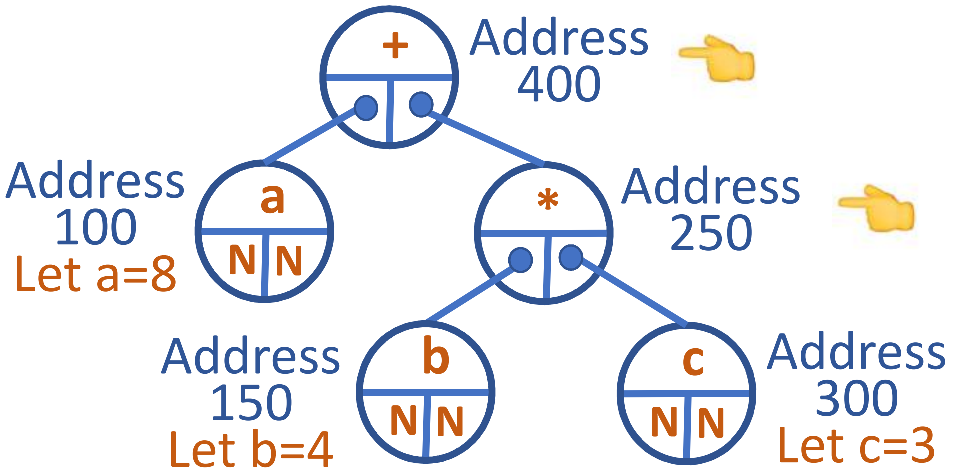
eval(150)
return 4

eval(300)
return 3

- Think in terms of recursion

eval(t) // 't' has the address of the root node of expression tree
  if t->data is an operator
    return eval (t->left) t->data eval(t->right)
  return t->data

Expression Tree Evaluation



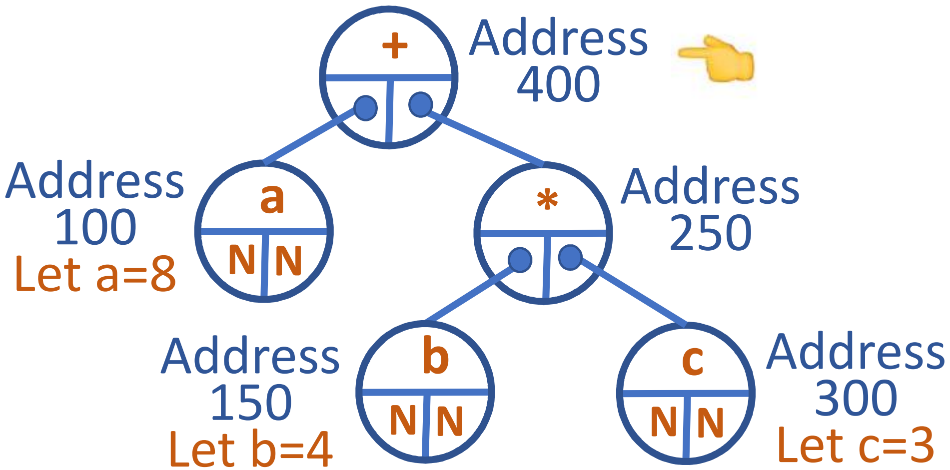
eval(400)
return 8 + eval(250) 12

eval(250)
return 12

- Think in terms of recursion

eval(t) // 't' has the address of the root node of expression tree
  if t->data is an operator
    return eval (t->left) t->data eval(t->right)
  return t->data

Expression Tree Evaluation



eval(400)

return 208 + 12

Postfix abc*+ : **20**

- Think in terms of recursion

eval(t) // 't' has the address of the root node of expression tree

if t->data is an operator

return eval (t->left) t->data eval(t->right)

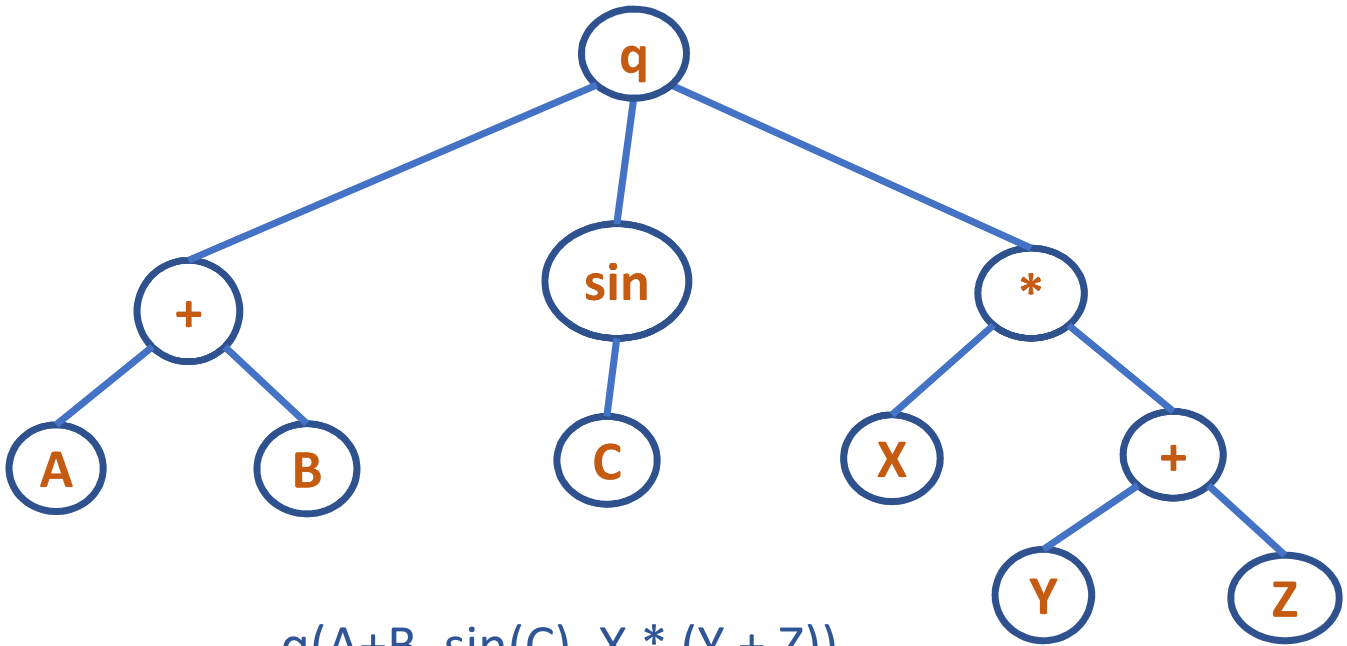
return t->data

General Expression Tree Evaluation

```
struct treenode
{
  short int utype;
   union{
      char operator[MAX];
       float val;
     }info;
   struct treenode *child;
   struct treenode *sibling;
};
typedef struct treenode TREENODE;
```

General Expression Tree Evaluation

Here node can be either an operand or an operator



q(A+B, sin(C), X * (Y + Z))

Tree representation of an arithmetic expression

General Expression Tree Evaluation

```
void replace(TREENODE *p)
{
  float val;
  TREENODE *q,*r;
  if(p->utype == operator)
  {
   q = p->child;
   while(q != NULL)
    {
      replace(q);
      q = q->next;}
    }
```

General Expression Tree Evaluation

```
  value = apply(p);
  p->utype = OPERAND;
  p->val = value;
  q = p->child;
  p->child = NULL;
  while(q != NULL)
  {
      r = q;
      q = q->next;
      free(r);
  }
 }
}
```

General Expression Tree Evaluation

```
float eval(TREENODE *p)
{
 replace(p);
 return(p->val);
 free(p);
}
```

Constructing a Tree

```
void setchildren(TREENODE *p,TREENODE *list)
{
   if(p == NULL) {
     printf("invalid insertion");
     exit(1);
   }
  if(p->child != NULL) {
    printf("invalid insertion");
    exit(1);
  }
  p->child = list;
}
```

Constructing a Tree

```
void addchild(TREENODE *p,int x)
{
 TREENODE *q;
 if(p==NULL)
 {
  printf("void insertion");
  exit(1);
 }
```

Constructing a Tree

```
r = NULL;
q = p->child;
while(q != NULL)
{
  r = q;
  q = q->next;
}
q = getnode();
q->info = x;
q->next = NULL;
```

Constructing a Tree

```
if(r==NULL)
   p->child=q;
 else
   r->next=q;
}
```

THANK YOU

Shylaja S S

Department of Computer Science
& Engineering

shylaja.sharath@pes.edu

+91 9449867804

DATA STRUCTURES AND ITS APPLICATIONS
UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Heap: Definition and Implementation

Shylaja S S

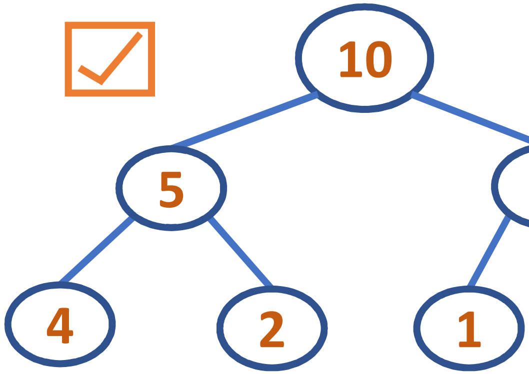
Department of Computer Science & Engineering

Heap Tree

Definition: A heap can be defined as a binary tree with keys assigned to its nodes (one key per node) provided the following two conditions are met:

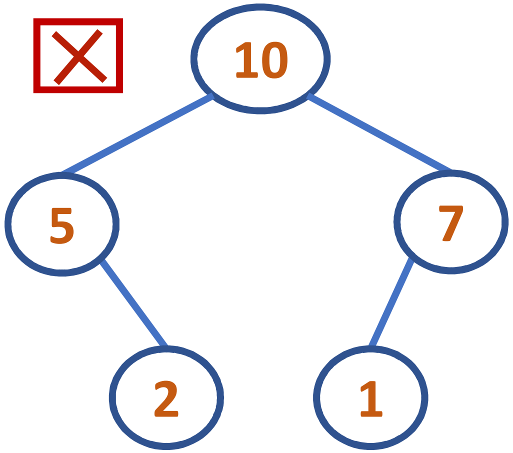
1. **The tree's shape requirement** - The binary tree is essentially complete, that is, all its levels are full except possibly the last level, where only some rightmost leaves may be missing
2. **The parental dominance requirement** - The key at each node is greater than or equal to the keys at its children. (This condition is considered automatically satisfied for all leaves.)

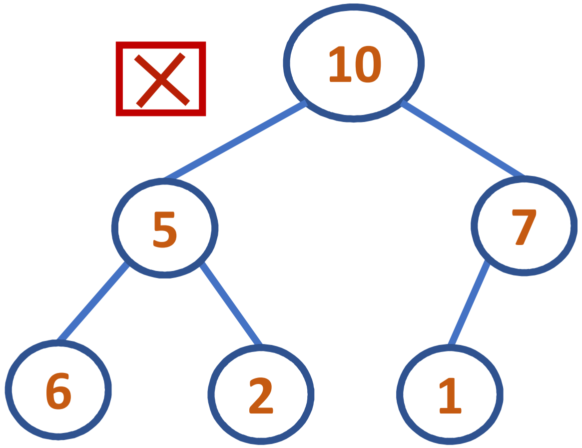
Heap Tree



Shape Requirement
not satisfied

Parental dominance
not satisfied





Only the topmost Binary Tree is a heap. Why?

Properties of Heap

1. There exists exactly one essentially complete binary tree with n nodes. Its height is equal to $\lfloor \log_2 n \rfloor$

2. The root of a heap always contains its largest element

3. A node of a heap considered with all its descendants is also a heap

4. A heap can be implemented as an array by recording its elements in the top-down, left-to-right fashion. It is convenient to store the heap's elements in positions 1 through n of such an array, leaving H[0] either unused or putting there a sentinel whose value is greater than every element in the heap.

Properties of Heap

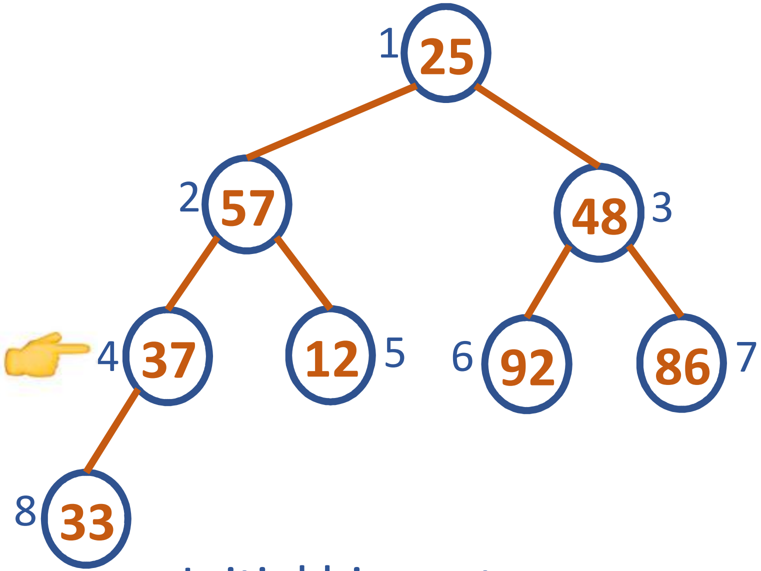
…

In such a representation,

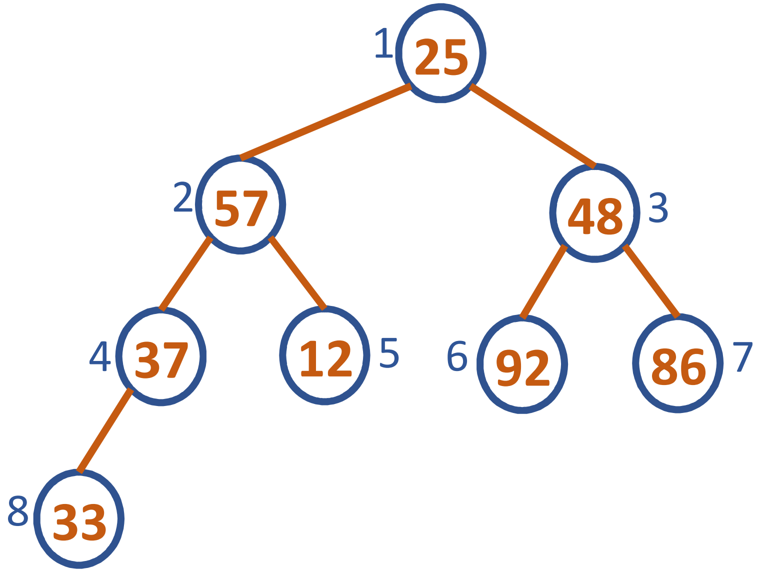
a) The parental node keys will be in the first ⌊n/2⌋ positions of the array, while the leaf keys will occupy the last ⌈n/2⌉ positions

b) The children of a key in the array's parental position i (1 <= i <= ⌊n/2⌋) will be in positions 2i and 2i + 1, and, correspondingly, the parent of a key in position i (2 <= i <= n) will be in position ⌊n/2⌋

Bottom Up Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33
Here, n=8



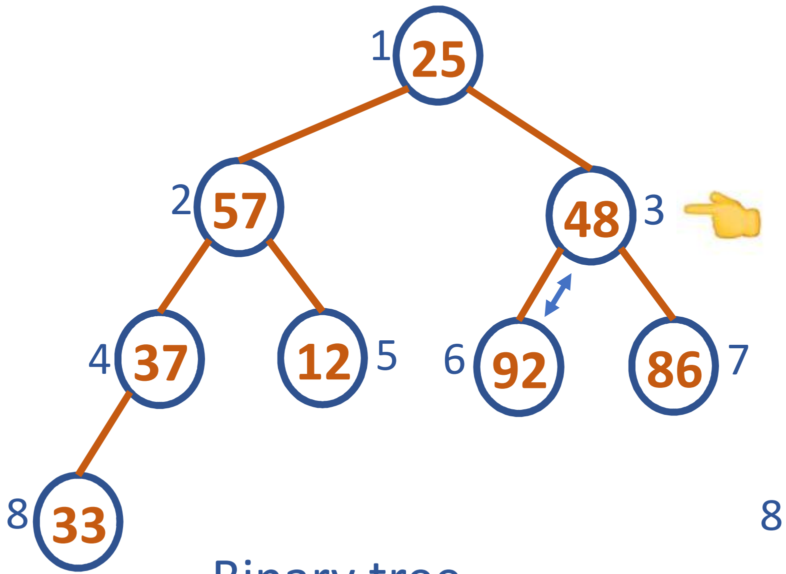
Initial binary tree



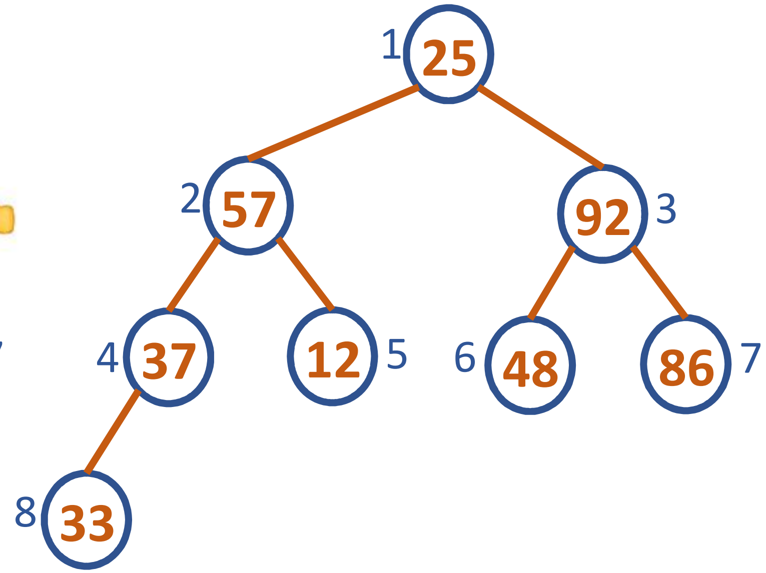
At k = 4, v = 37
Compare 37 with its only child 33
37 > 33, it’s a heap at k=4

Heap Construction – Bottom Up

Bottom Up Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33
Here, n=8



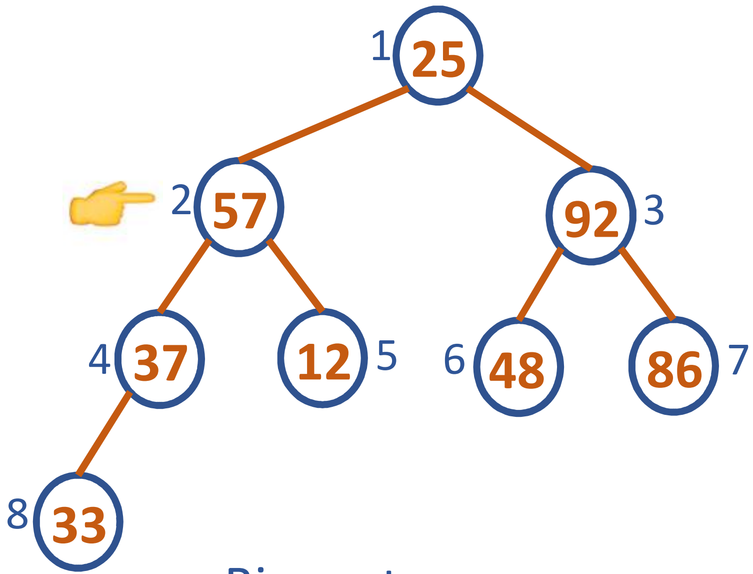
Binary tree
after one iteration at k=4

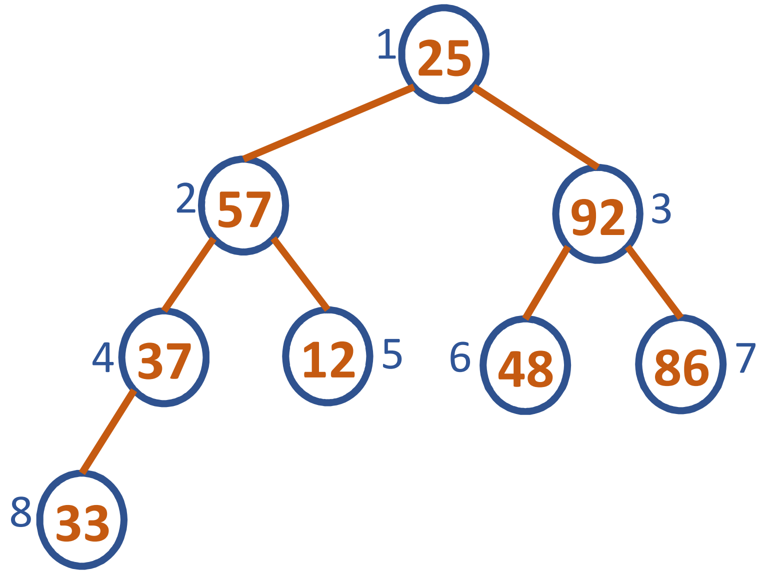


At k = 3, v = 48
Largest child: 92
Compare 48 with its largest child
48 < 92, Heapify

Heap Construction – Bottom Up

Bottom Up Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33
Here, n=8





Binary tree
after two iterations at k=4, k=3

At k = 2, v = 57

Largest child: 37

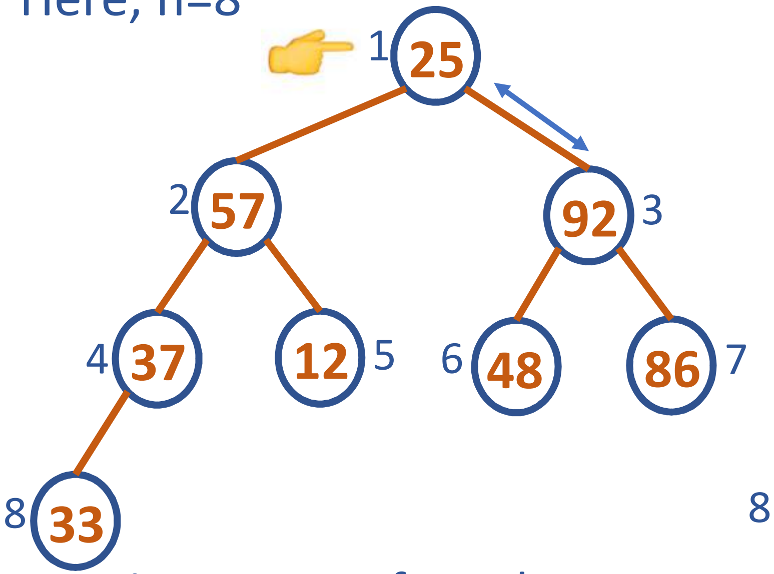
Compare 57 with its largest child

57 > 37, it’s a heap at k=2

Heap Construction – Bottom Up

Bottom Up Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33

Here, n=8



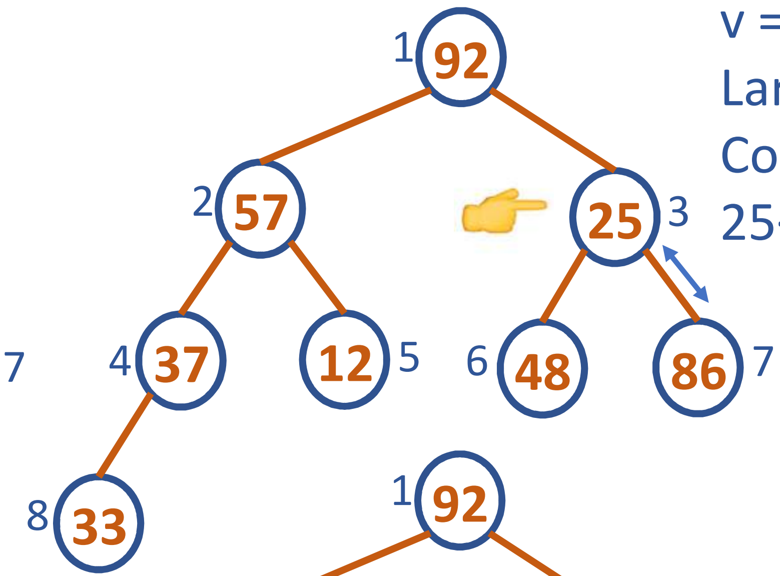
Binary tree after three iterations at k=4, k=3, k=2

At k = 1, v = 25

Largest child: 92

Compare 25 with its largest child

25 < 92, Heapify

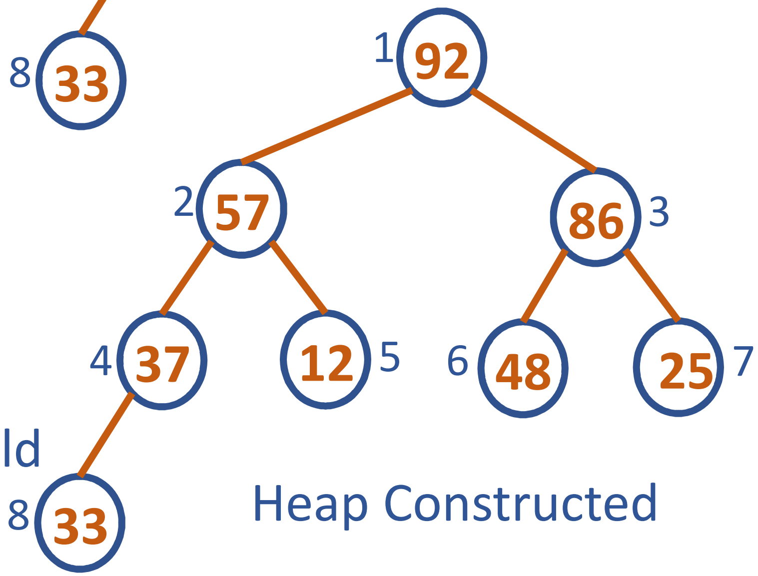


v = 25, Now at k = 3,

Largest child: 86

Compare 25 with its largest child

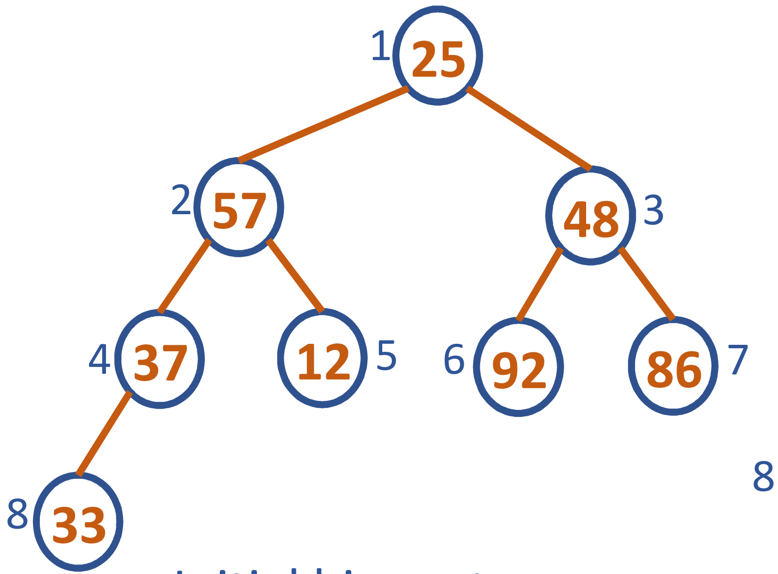
25<86, Heapify



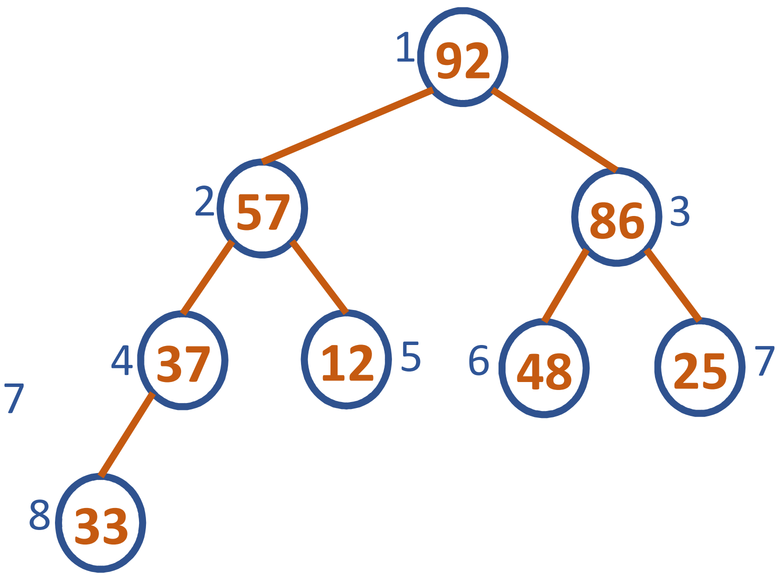
Heap Constructed

Heap Construction – Bottom Up

Bottom Up Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33
Here, n=8



Initial binary tree



Bottom Up Heap Constructed

Heap Construction – Bottom Up

```
ALGORITHM HeapBottomUp(H[1…n])
//Constructs a heap from the elements of a given array by bottom-up algorithm
//Input: An array H[1…n] of orderable items
//Output: A heap H[1…n]
for i ← ⌊n/2⌋ downto 1 {
    k ← i
    v ← H[k]
    heap ← false
    while not heap and 2*k ≤ n {
        j ← 2*k
        if j < n                        //if there are two children
           if H[j] < H[j+1]
                j ← j+1                 //find position of largest child
        if v ≥ H[j]            //if key of parent node ≥ key of largest child
           heap ← true                  //it's a heap
        else {                          //heapify
                H[k] ← H[j]
                k ← j
        }       //end of else
    }   //end of while
    H[k] ← v
}   //end of for
```

Efficiency

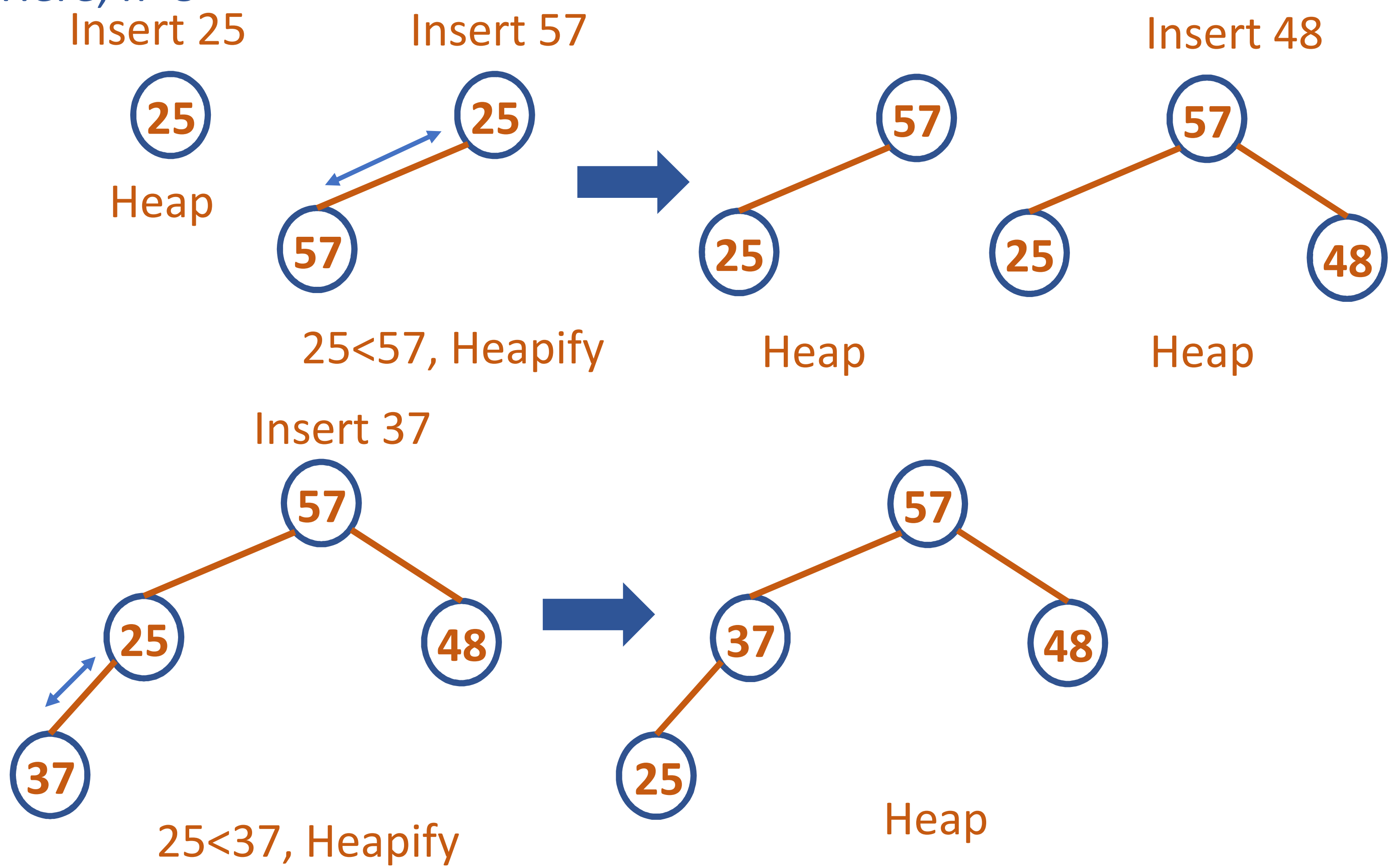
$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{level\ i\ keys} 2(h-i)$$

$$= \sum_{i=0}^{h-1} 2(h-i)2^i$$

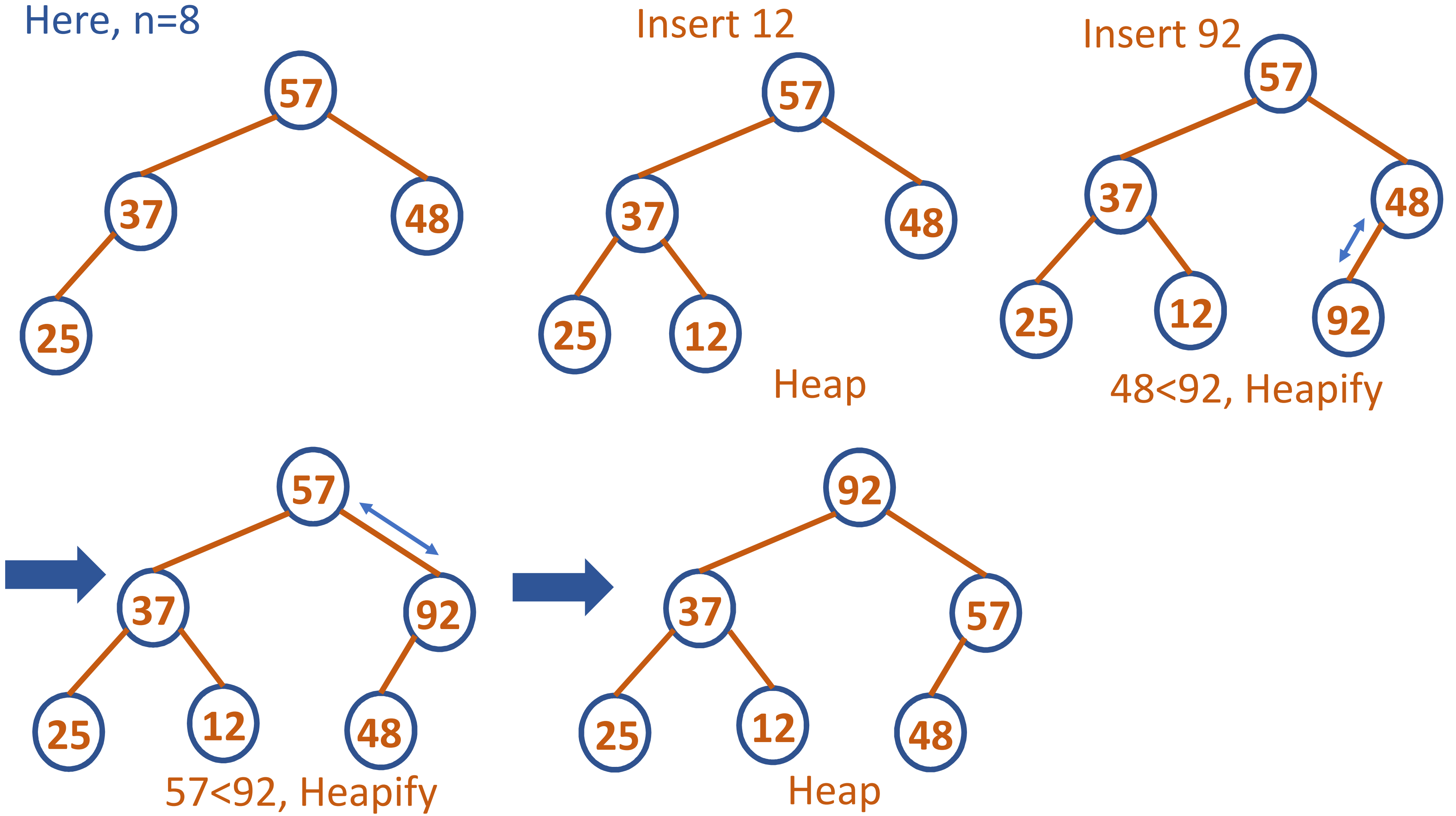
$$= 2(n - \log_2(n+1))$$

Heap Construction – Top Down

Top Down Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33
Here, n=8



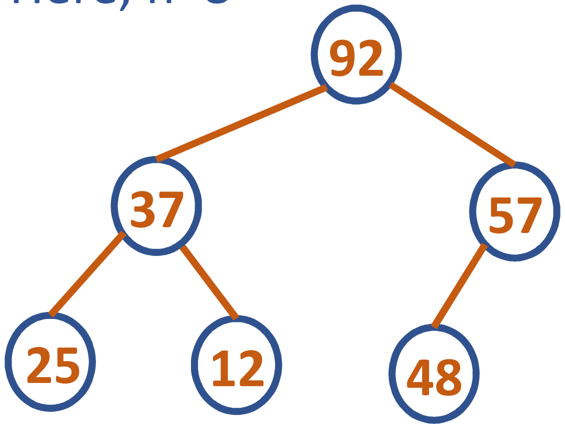
Top Down Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33
Here, n=8



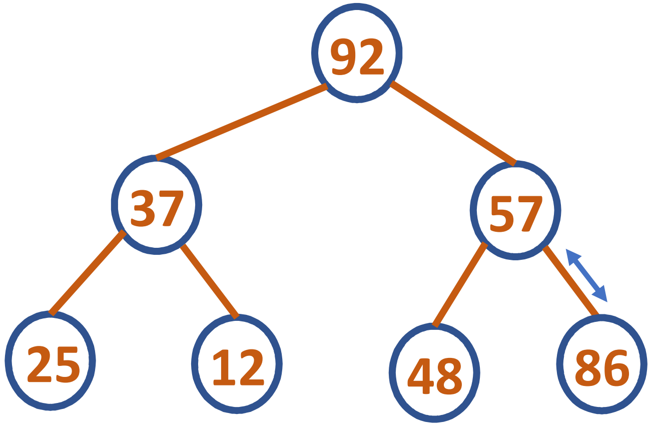
Heap Construction – Top Down

Top Down Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33

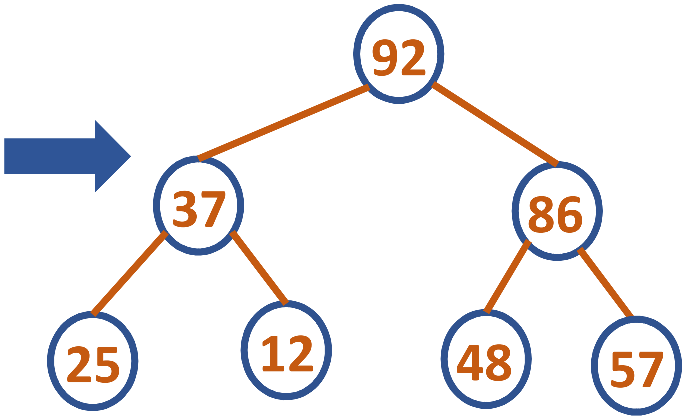
Here, n=8



Insert 86



57<86, Heapify

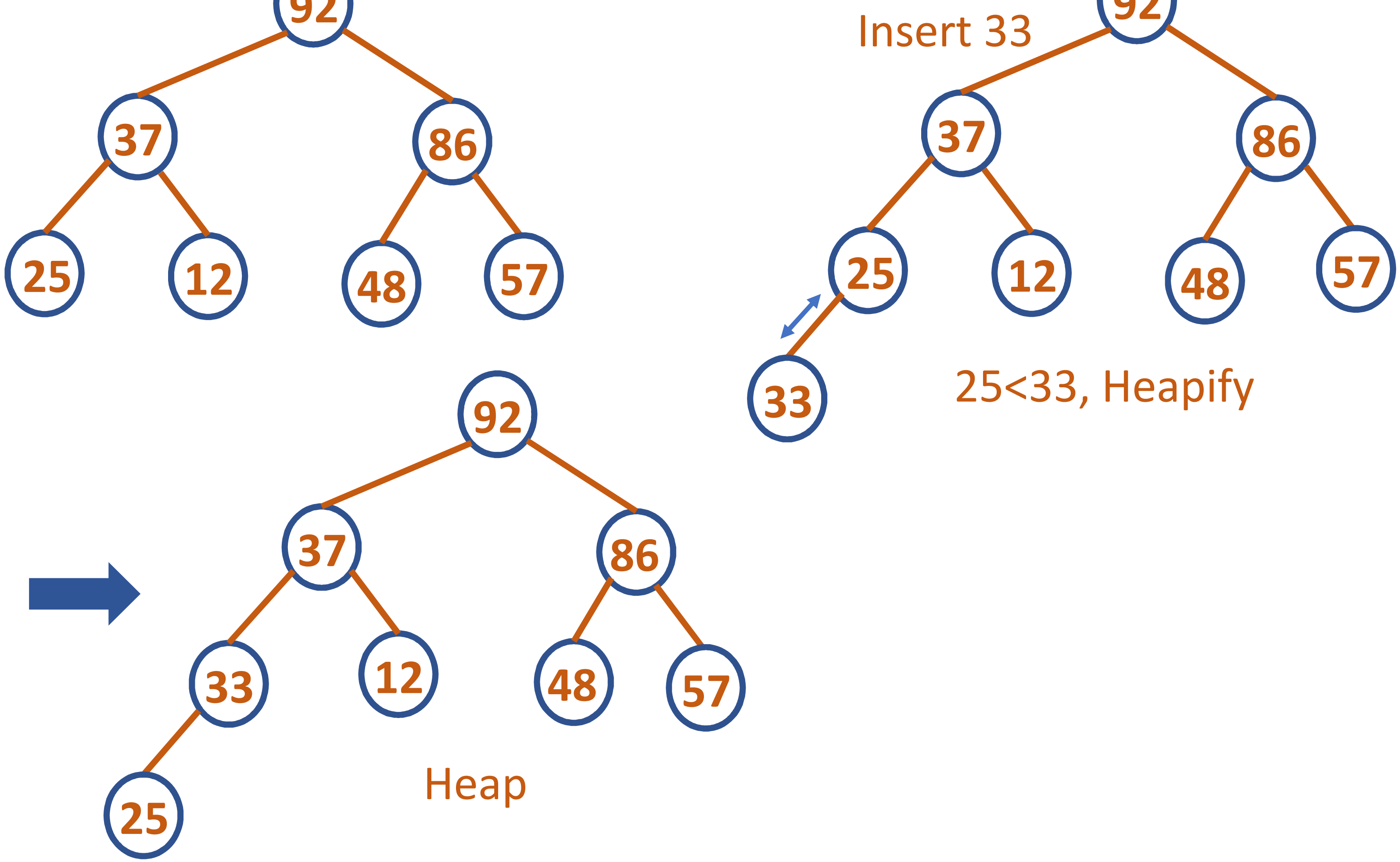


Heap

Heap Construction – Top Down

Top Down Heap Construction: 25, 57, 48, 37, 12, 92, 86, 33

Here, n=8



Heap Construction – Top Down

1. First, attach a new node with key *K* in it after the last leaf of the existing heap
2. Then sift *K* up to its appropriate place in the new heap as follows
3. Compare *K* with its parent's key: if the latter is greater than or equal to *K,* stop (the structure is a heap);
4. otherwise, swap these two keys and compare *K* with its new parent
5. This swapping continues until *K* is not greater than its last parent or it reaches the root
6. In this algorithm, too, we can sift up an empty node until it reaches its proper position, where it will get *K* 's value

Heap Construction – Top Down

- Efficiency of insertion is O(log n)

THANK YOU

Shylaja S S

Department of Computer Science
& Engineering

shylaja.sharath@pes.edu

+91 9449867804

DATA STRUCTURES AND ITS APPLICATIONS
UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES AND ITS APPLICATIONS

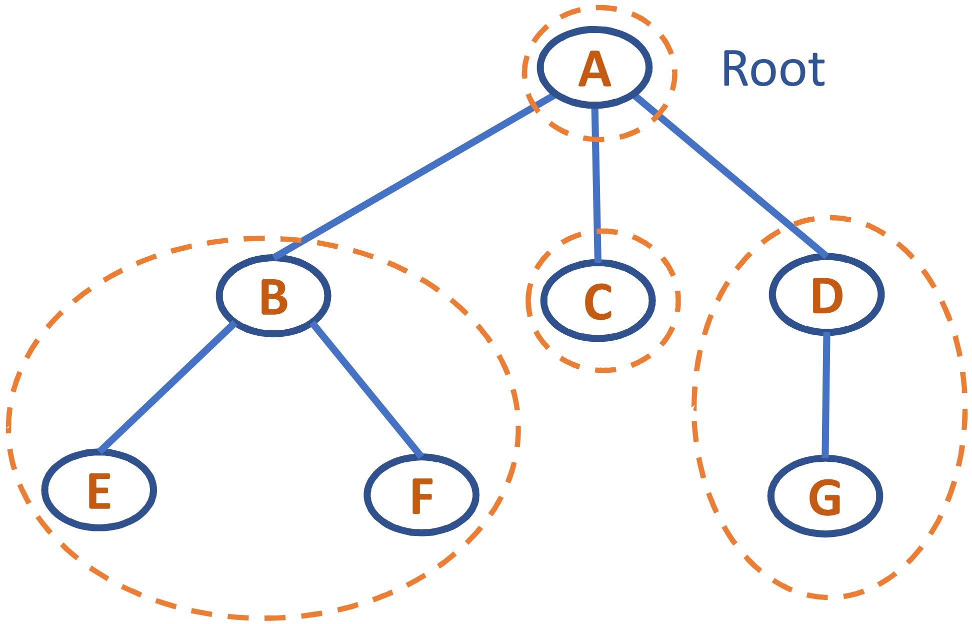
Basic Concept and Definitions: Trees

Shylaja S S
Department of Computer Science & Engineering

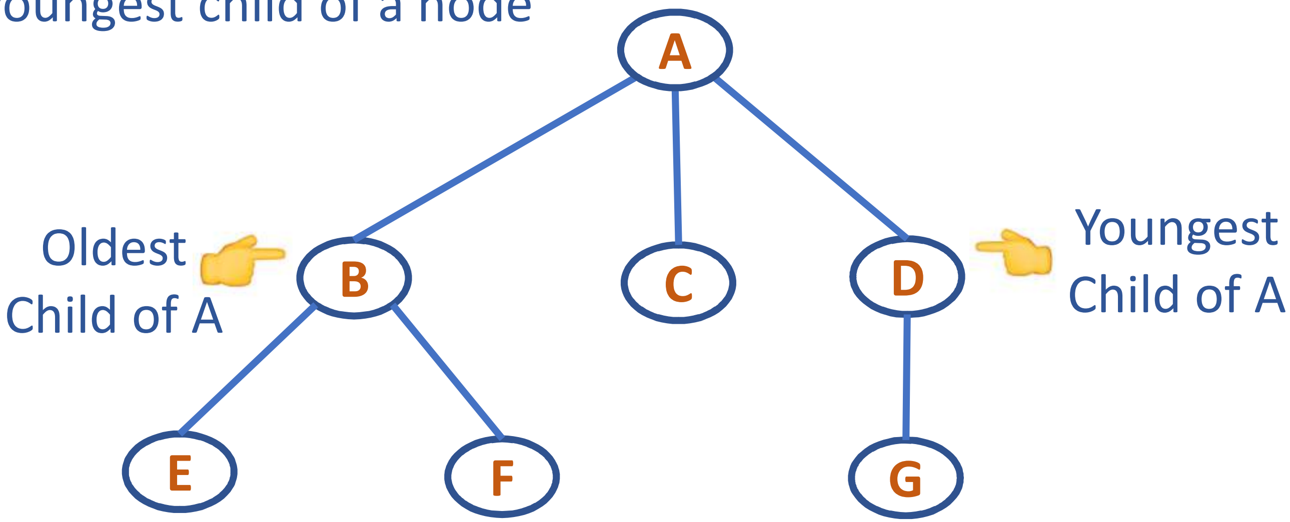
DATA STRUCTURES AND ITS APPLICATIONS

Trees

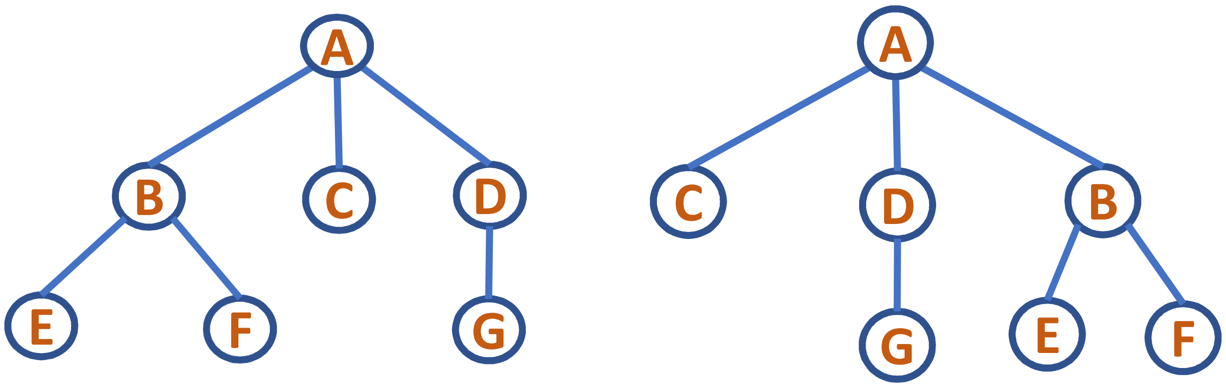
- Non Linear Data Structure
- Finite nonempty set of elements
  - One element is the root
  - Remaining elements are partitioned into m≥0 disjoint subsets each of which is itself a tree



- Ordered Tree: a tree in which subtrees of each node forms an ordered set
- In such a tree we define first, second, ..., Last child of a particular node
- First child is called the oldest child and the last child the youngest child of a node



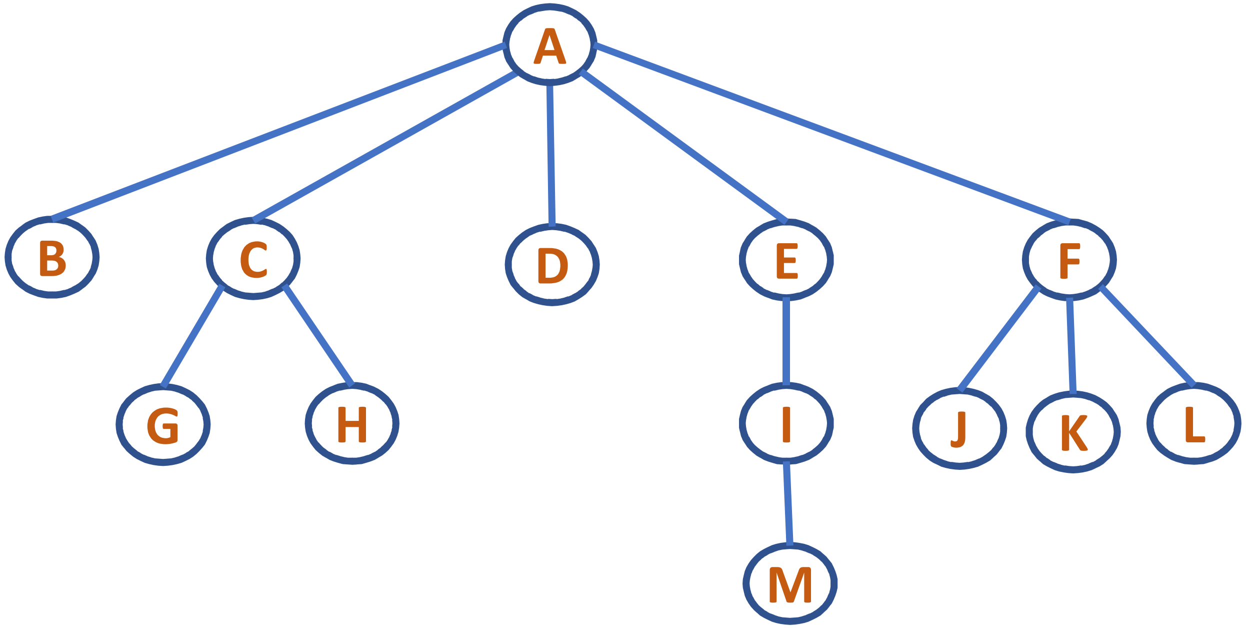
As unordered trees the below figures are equivalent but as ordered trees, they are different



N-ary Tree & Forest

n-ary tree: A rooted tree in which each node has no more than **n** children

- A binary tree is an n-ary tree with n=2
- n-ary tree with n=5



- Forest: is an ordered set of ordered trees

Representation of trees:

Tree node options:

```
struct treenode{
        int info;
        struct treenode *child[MAX];
};
```

where MAX is a constant

Restrictions with the above implementation:

- A node cannot have more than MAX children. Therefore cannot expand the tree

2nd implementation:

- All the children of a given node are linked and only the oldest child is linked to the parent
- A node has link to first child and a link to immediate sibling

```
struct treenode{
        int info;
        struct treenode *child;
        struct treenode *sibling;
};
```

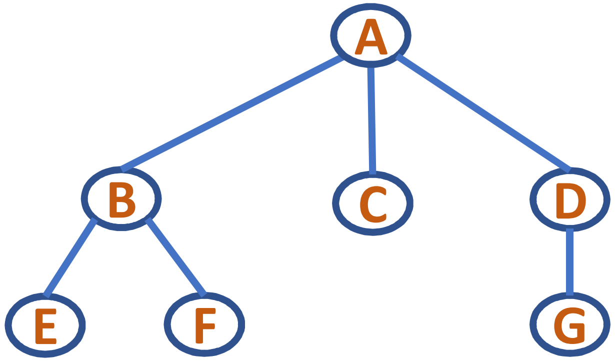
Conversion of an n-ary Tree to a Binary Tree

Left Child – Right Sibling Representation

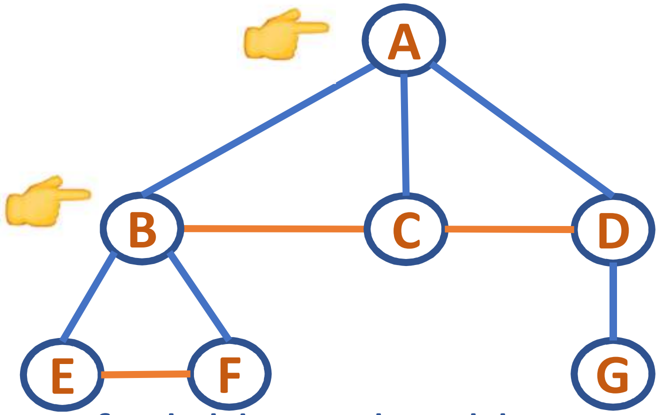
- Link all the siblings of a node
- Delete all links from a node to its children except for the link to its leftmost child
- The left child in binary tree is the node which is the oldest child of the given node in an n-ary tree, and the right child is the node to the immediate right of the given node on the same horizontal line. Such a binary tree will not have a right sub tree
- The node structure corresponds to that of

Data	
Left Child	Right Sibling

Conversion of an n-ary Tree to a Binary Tree



3-ary tree

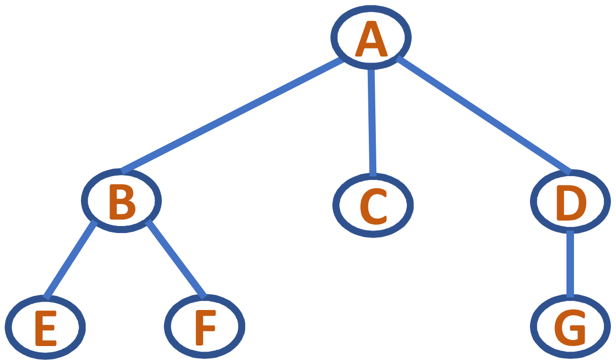


Left child – Right sibling
Representation of the above 3-ary tree

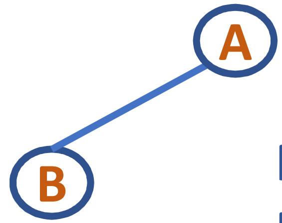
Link all siblings of a Node

Delete all links from a Node to its children except for the link to its leftmost child

Conversion of an n-ary Tree to a Binary Tree

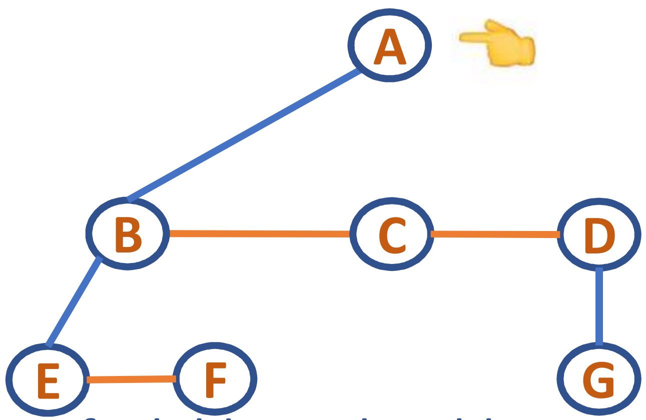


3-ary tree



Node Below is B – Left child

No Right sibling so – no Right child

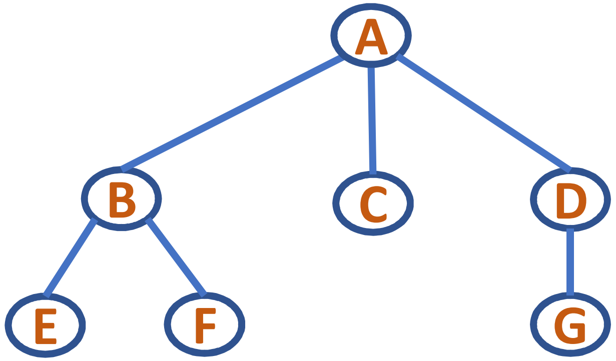


Left child – Right sibling

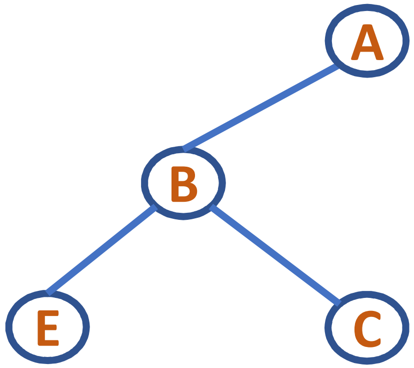
Representation of the above 3-ary tree

Corresponding binary tree

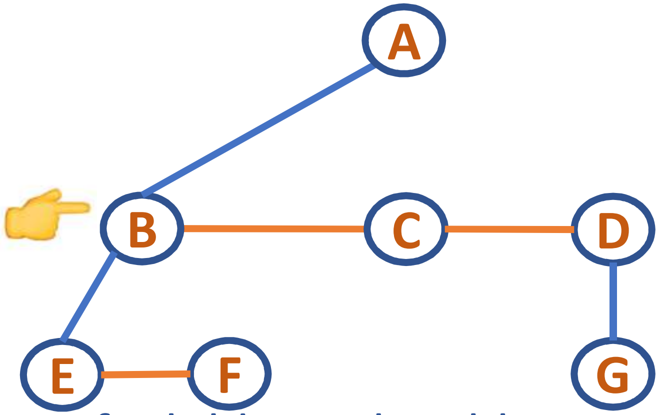
Conversion of an n-ary Tree to a Binary Tree



3-ary tree



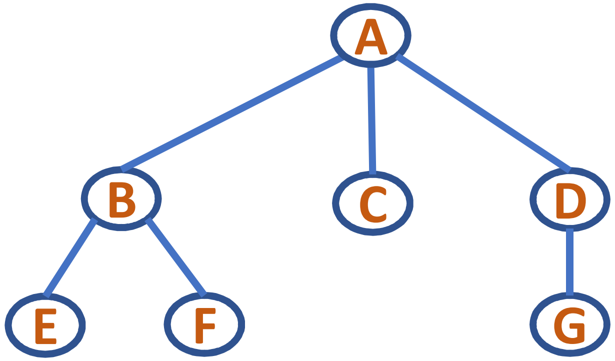
Node Below is E – Left child
Next Right sibling is C – Right child



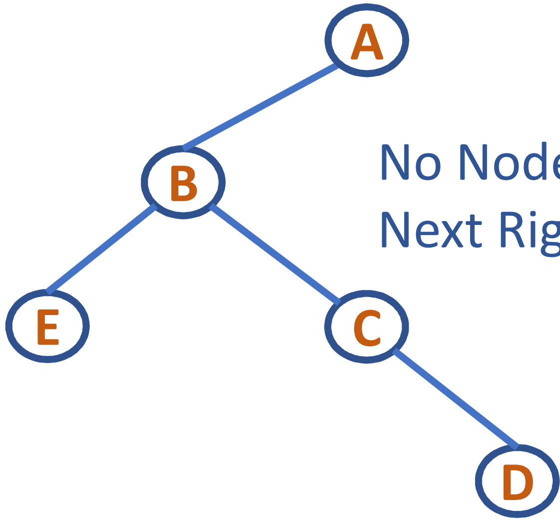
Left child – Right sibling
Representation of the above 3-ary tree

Corresponding
binary tree

Conversion of an n-ary Tree to a Binary Tree

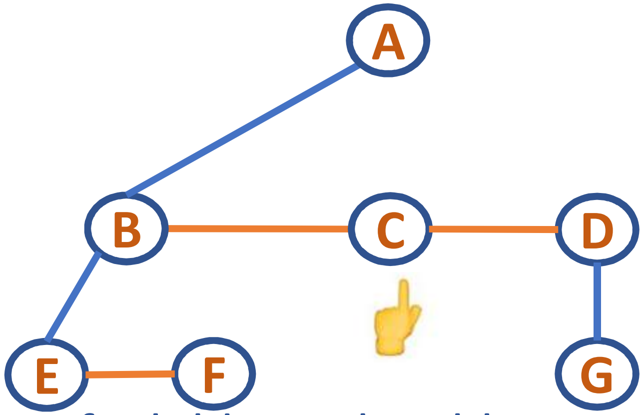


3-ary tree



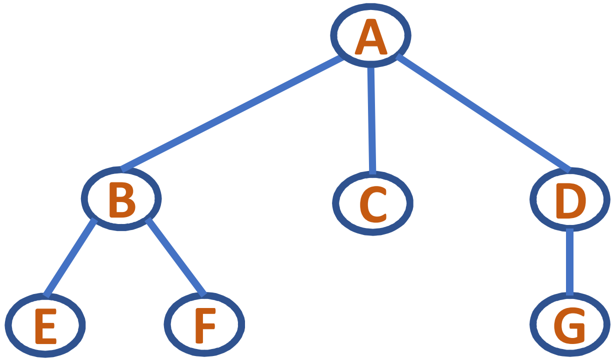
No Node Below so – no Left child
Next Right sibling is D – Right child

Corresponding
binary tree

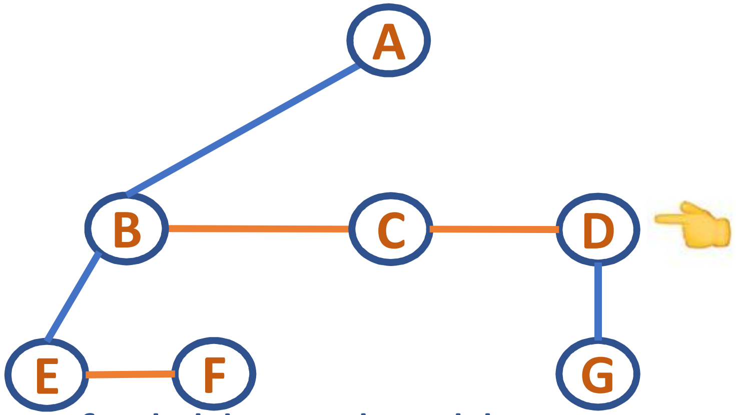


Left child – Right sibling
Representation of the above 3-ary tree

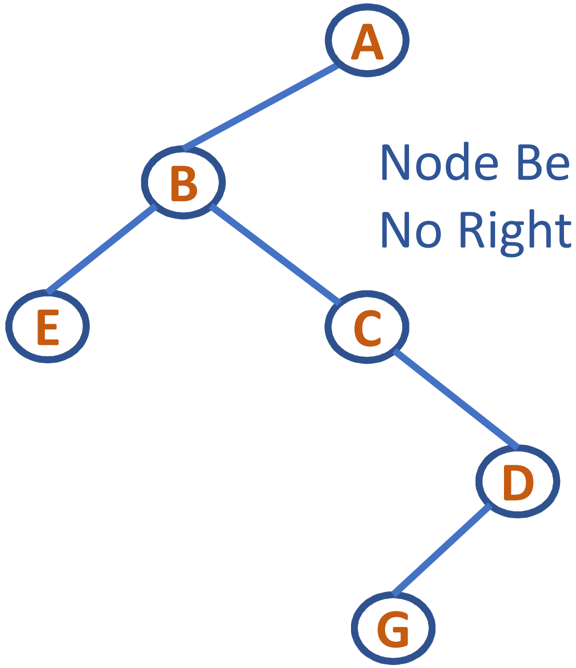
Conversion of an n-ary Tree to a Binary Tree



3-ary tree



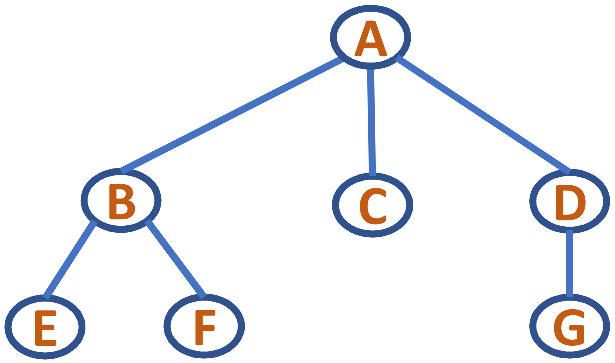
Left child – Right sibling
Representation of the above 3-ary tree



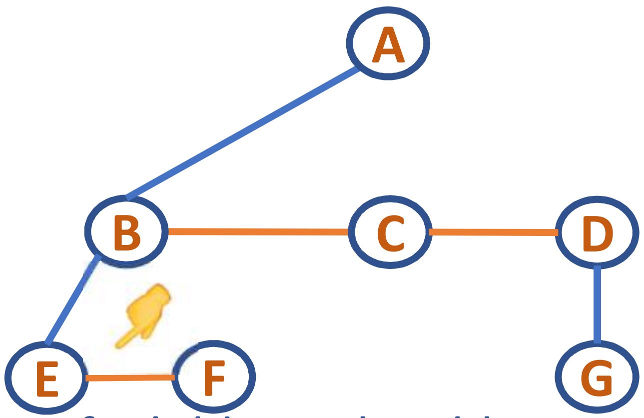
Node Below is G – Left child
No Right sibling so – no Right child

Corresponding
binary tree

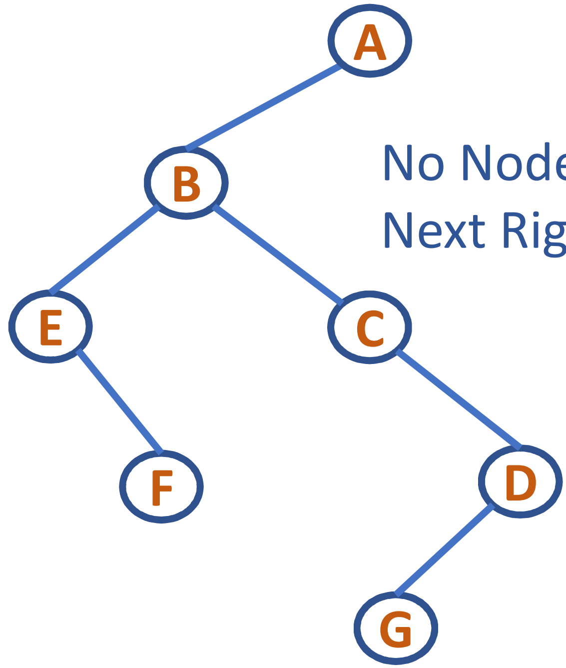
Conversion of an n-ary Tree to a Binary Tree



3-ary tree



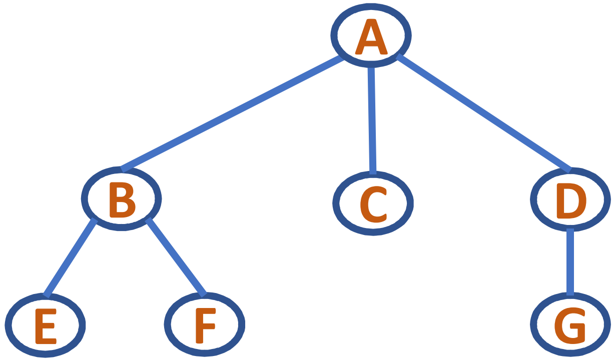
Left child – Right sibling
Representation of the above 3-ary tree



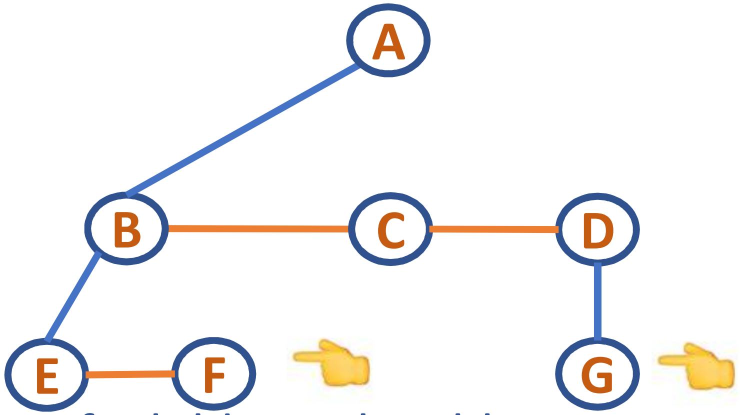
No Node Below so – no Left child
Next Right sibling is F – Right child

Corresponding
binary tree

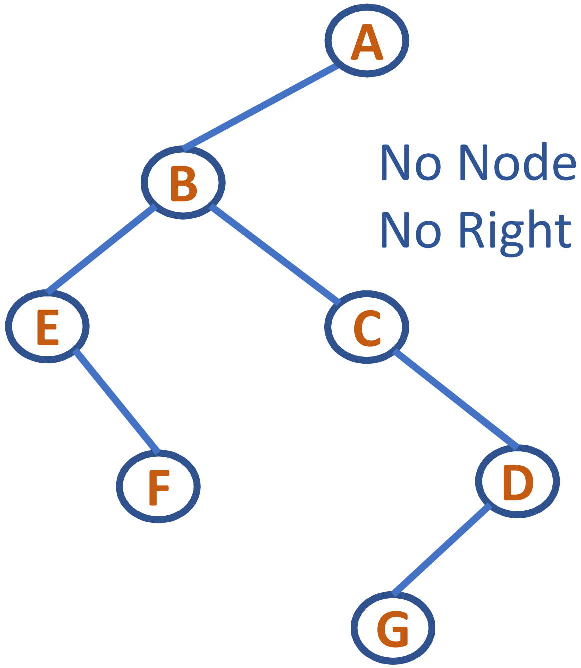
Conversion of an n-ary Tree to a Binary Tree



3-ary tree



Left child – Right sibling
Representation of the above 3-ary tree



No Node Below so – no Left child
No Right sibling so – no Right child

Corresponding
binary tree

Conversion of a Forest to a Binary Tree

- Right Child of the root node of every resulting binary tree will be empty. This is because the root of the tree we are transforming has no siblings.

- On the other hand, if we have a forest then these can all be transformed into a single binary tree as follows:
    - First obtain the binary tree representation of each of the trees in the forest
    - Link all the binary trees together through the right sibling field of the root nodes

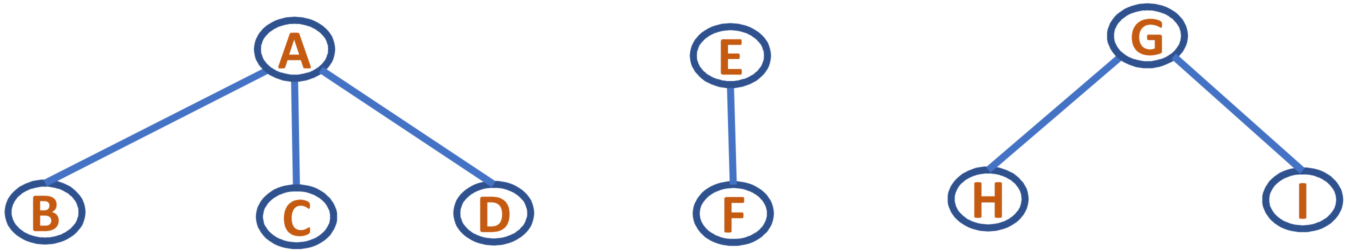
Conversion of a Forest to a Binary Tree

Conversion of a Forest to a Binary Tree can be formally defined as follows:

- If $T_1,...,T_n$ is a forest of n trees, then the binary tree corresponding to this forest, denoted by $B(T_{1,} ..., T_n)$:
  - is empty if $n = 0$
  - has root equal to root (T_1)
  - has left subtree equal to $B(T_{11}, T_{12}, ...,T_{1m})$ where $T_{11} ,..., T_{1m}$ are the subtrees of root(T_1)
  - has right subtree $B(T_2, ..., T_n)$

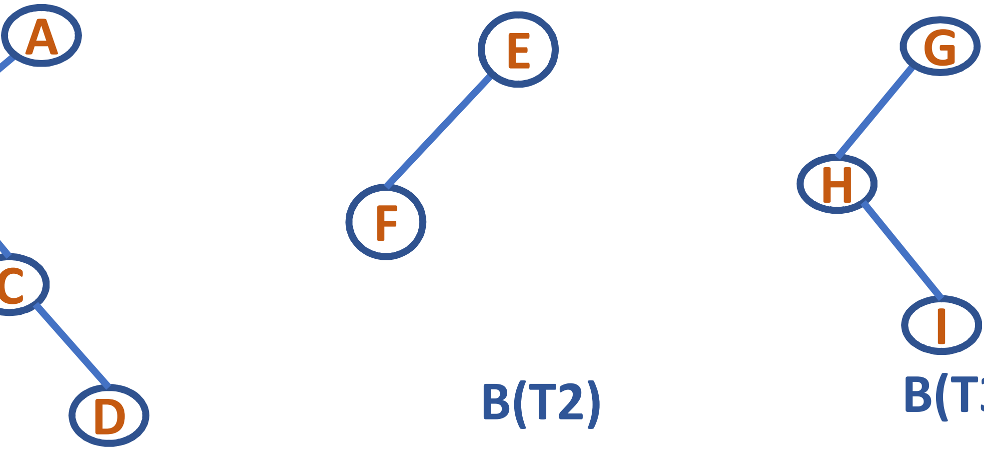
Conversion of a Forest to a Binary Tree

Consider the following Forest with three Trees



T1 T2 T3

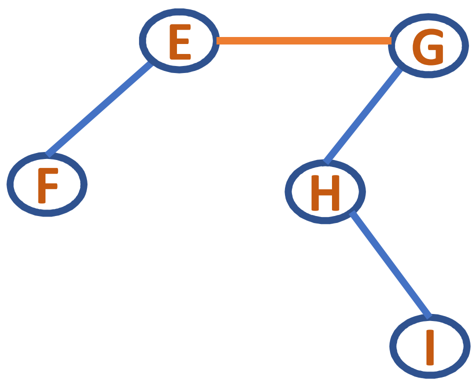
Corresponding Binary Trees



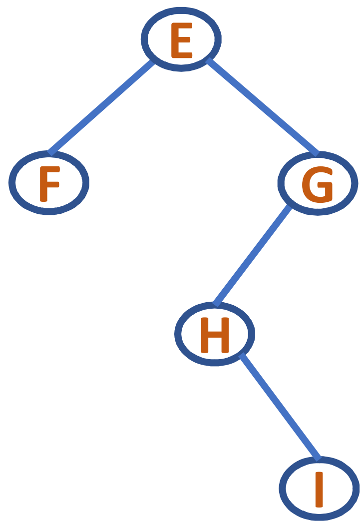
B(T1) B(T2) B(T3)

Conversion of a Forest to a Binary Tree

Link B(T2) and B(T3)



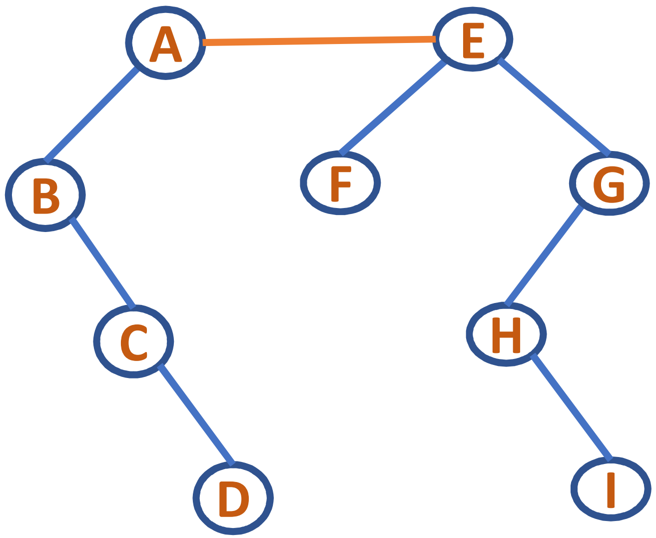
G becomes Right Child of E



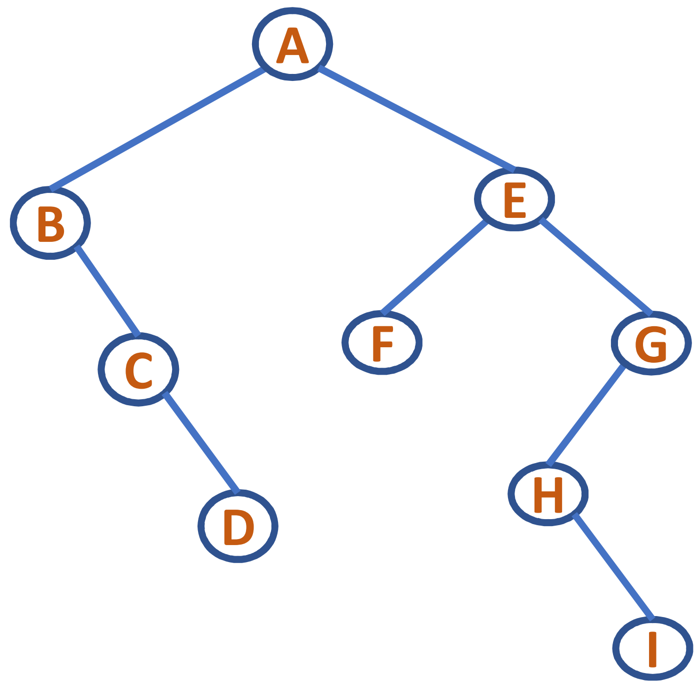
Corresponding Binary Tree
B(T2, T3)

Conversion of a Forest to a Binary Tree

Link B(T1) and B(T2, T3)



E becomes Right Child of A



Corresponding Binary Tree B(T1, T2, T3)

THANK YOU

Shylaja S S

Department of Computer Science
& Engineering

shylaja.sharath@pes.edu

+91 9449867804

DATA STRUCTURES AND ITS APPLICATIONS
UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES AND ITS APPLICATIONS

n-ary Tree Traversal

Shylaja S S
Department of Computer Science & Engineering

Structure of a treenode revisited

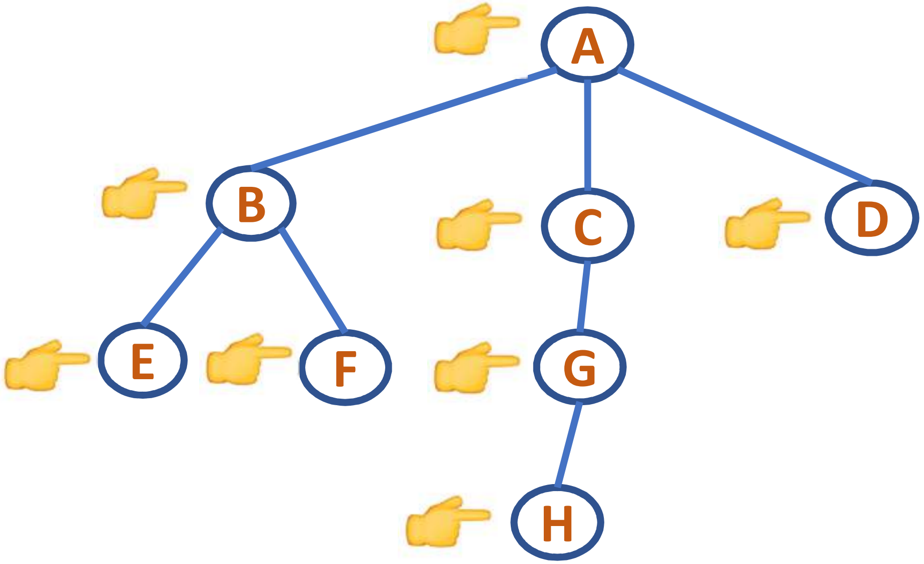
```
struct treenode{
        int info;
        struct treenode *child;
        struct treenode *sibling;
};
```

With the treenode implemented as having pointers to first child and immediate sibling, the traversal preorder, inorder and postorder for a tree are defined as follows:

Preorder:

1. Visit the root of the first tree in the forest
2. Traverse in preorder the forest formed by the subtrees of the first tree, if any
3. Traverse in preorder the forest formed by the remaining trees in the forest, if any

Preorder Tree Traversal



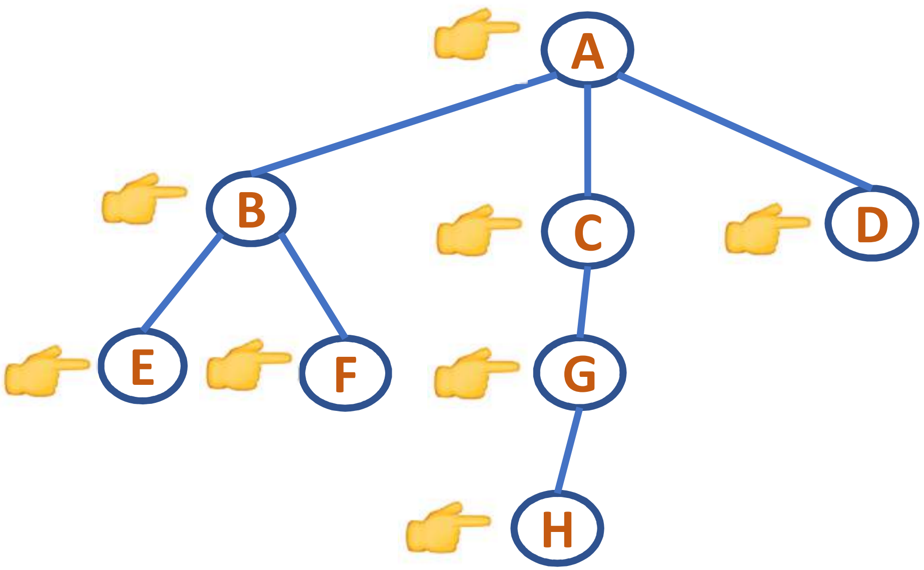
A B E F C G H D

```
void preorder(TREE *root)
{
    if(root!=NULL)
    {
      printf(" %d ",root->info);
      preorder(root->child);
      preorder(root->sibling);
    }
}
```

Inorder

1. Traverse in inorder the forest formed by the subtrees of the first tree, if any
2. Visit the root of the first tree in the forest
3. Traverse in inorder the forest formed by the remaining trees in the forest, if any

Inorder Tree Traversal



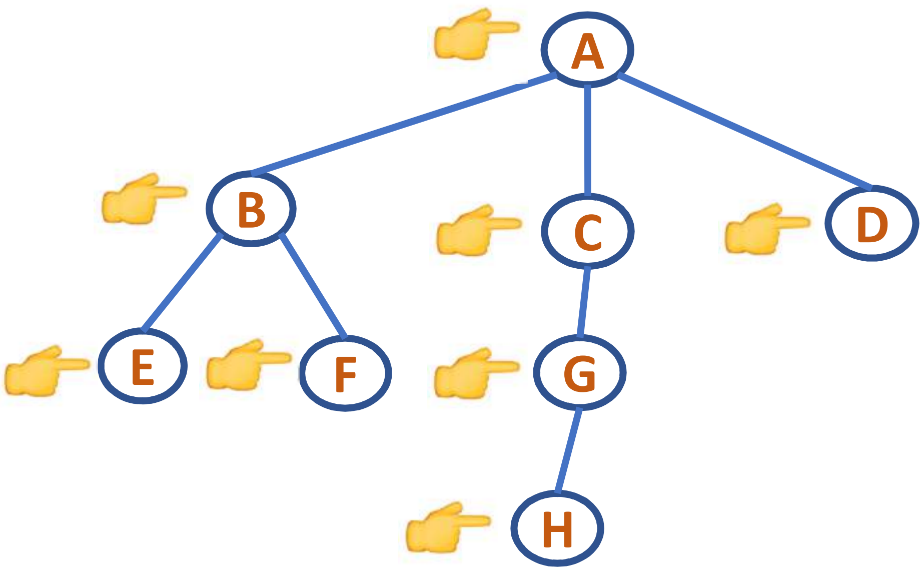
E F B H G C D A

```
void inorder(TREE *root)
{
  if(root!=NULL)
   {
     inorder(root->child);
     printf(" %d ",root->info);
     inorder(root->sibling);
   }
}
```

Postorder

1. Traverse in postorder the forest formed by the subtrees of the first tree, if any
2. Traverse in postorder the forest formed by the remaining trees in the forest, if any
3. Visit the root of the first tree in the forest

Postorder Tree Traversal



F E H G D C B A

```
void postorder(TREE *root)
{
  if(root!=NULL)
   {
     postorder(root->child);
     postorder(root->sibling);
     printf(" %d ", root->info);
   }
}
```

THANK YOU

Shylaja S S

Department of Computer Science
& Engineering

shylaja.sharath@pes.edu

DATA STRUCTURES AND ITS APPLICATIONS
UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Implementation of Priority Queue using min heap/max heap

Shylaja S S

Department of Computer Science & Engineering

Ascending and Descending Heap

- Ascending Heap: Root will have the lowest element. Each node's data is greater than or equal to its parent's data. It is also called min heap.

- Descending Heap: Root will have the highest element. Each node's data is lesser than or equal to its parent's data. It is also called max heap.

Priority Queue using Heap

- Priority Queue is a Data Structure in which intrinsic ordering of the elements does determine the results of its basic operations

- Ascending Priority Queue: is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed

- Descending Priority Queue: is a collection of items into which items can be inserted arbitrarily and from which only the largest item can be removed

Priority Queue using Heap

Consider the properties of a heap

- The entry with largest key is on the top(Descending heap) and can be removed immediately. But O(logn) time is required to readjust the heap with remaining keys

- If another entry need to be done, it requires O(logn)

- Therefore, heap is advantageous to implement a Priority Queue

Priority Queue using Heap: Implementation

- dpq: Array that implements descending heap of size k (position from 0 to k-1)

- pqinsert(dpq,k,elt): insert element into the heap dpq of size k. Size increases to k+1

- This insertion is done using siftup operation

Algorithm for siftup

```
c = k;
p = (c-1)/2;
while(c>0 && dpq[p]<elt) {
   dpq[c]=dpq[p];
   c=p;
   p=(c-1)/2;
}
dpq[c]=elt;
```

Priority Queue using Heap: Implementation

pqmaxdelete(dpq,k) //for a descending heap of size k

p = dpq[0];

adjustheap(0,k-1)

return p;

Algorithm largechild(p,m)

```
c = 2*p+1;
if(c+1 <= m && x[c] < x[c+1])
  c=c+1;
if(c > m)
  return -1;
else
  return (c);
```

Priority Queue using Heap: Implementation

Algorithm adjustheap(root,k) //recursive

```
p = root;
c = largechild(p,k-1);
if(c >= 0 && dpq[k] < dpq[c]){
  dpq[p] = dpq[c];
  adjustheap(c,k);
}
else
  dpq[p] = dpq[k];
```

Iterative version

```
p = root;
kvalue = dpq[k];
c = largechild(p,k-1);
while(c >= 0 && kvalue < dpq[c]){
    dpq[p] = dpq[c];
    p = c;
    c = largechild(p,k-1);
}
dpq[p] = kvalue;
```

THANK YOU

Shylaja S S

Department of Computer Science
& Engineering

shylaja.sharath@pes.edu

+91 9449867804

DATA STRUCTURES AND ITS APPLICATIONS
UE19CS202

Shylaja S S & Kusuma K V
Department of Computer Science
& Engineering

DATA STRUCTURES AND ITS APPLICATIONS

Programs on Binary Trees

Shylaja S S

Department of Computer Science & Engineering

```
//Returns the smallest element in Binary Search Tree
int minimum(struct tnode *t)
{
  while(t->left!=NULL)
   t=t->left;
  return(t->data);
}
```

```
//Returns the largest element in Binary Search Tree
int maximum(struct tnode *t)
{
  while(t->right!=NULL)
   t=t->right;
  return(t->data);
}
```

```
//Computes the height of a Binary Tree
int height(struct tnode *t)
 {
  if(t==NULL)
    return -1;
  if((t->left==NULL)&&(t->right==NULL))
    return 0;
  return (1+max(height(r->left),height(r->right)));
 }
```

```
//Count the number of leaf nodes in a Binary Tree
int leafcount(struct tnode *t)
{
   if(t==NULL)
    return 0;
  if((t->left==NULL)&&(t->right==NULL))
    return 1;
  int l=leafcount(t->left);
  int r=leafcount(t->right);
  return(l+r);
}
```

THANK YOU

Shylaja S S

Department of Computer Science
& Engineering

shylaja.sharath@pes.edu

+91 9449867804