SIM GLOBAL EDUCATION

UNIVERSITY OF WOLLONGONG AUSTRALIA

CSCI203 - Algorithms and Data Structures

Tree (1)

Sionggo Japit

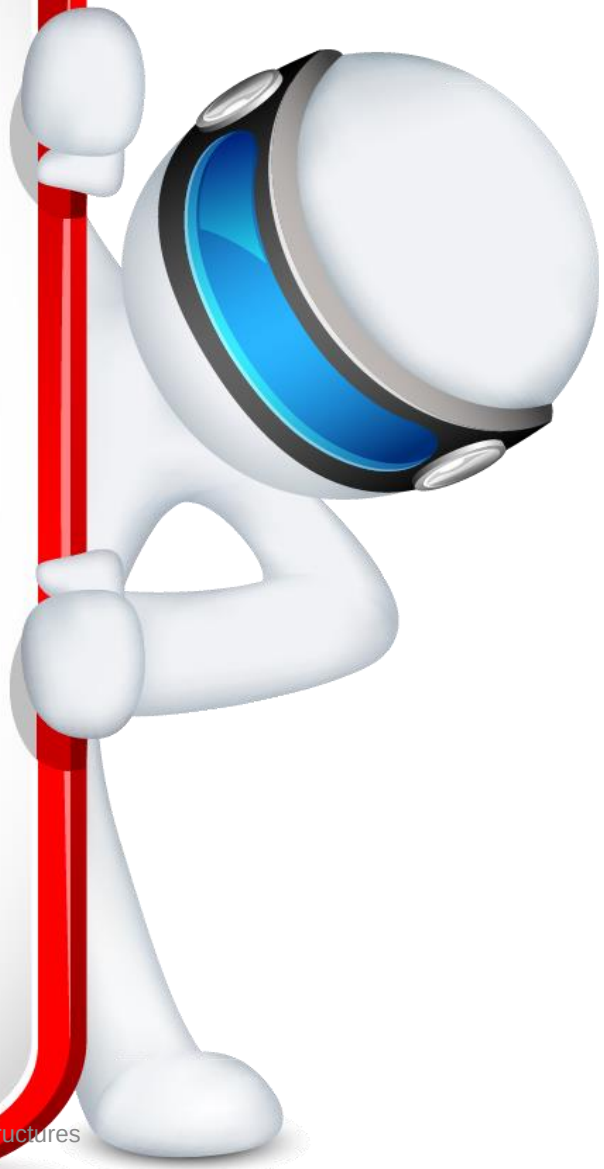
sjapit@uow.edu.au

2 January 2023

Outline

- Binary tree traversal
- Binary tree operation
  - Search
  - Insert
  - Delete
- AVL trees

Binary Tree Traversal




Binary Tree
Traversal

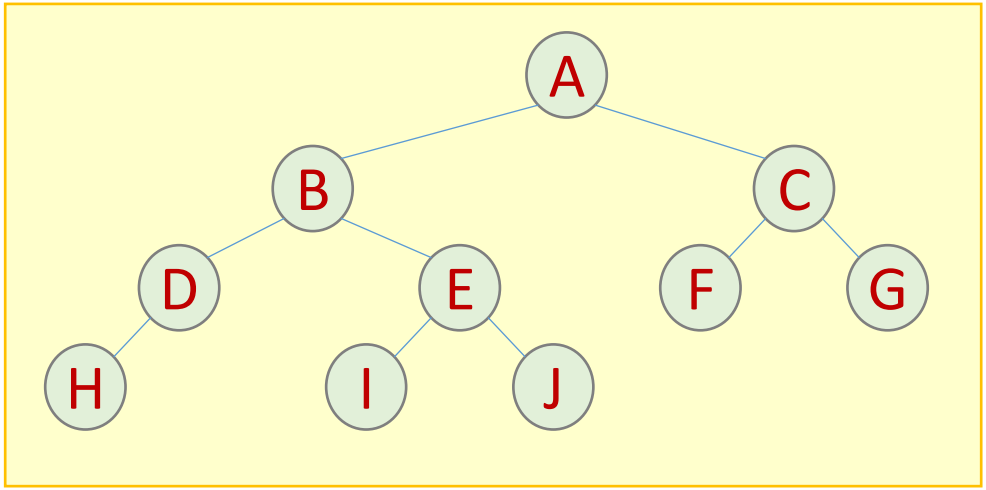
Binary Tree Traversal

- To traverse a binary tree is to “visit” each node in some prescribed order
  - “Visit” – For example, visit to print the data at each node
- Three most common traversal order
  - Preorder
  - Inorder
  - Postorder

Preorder Traversal

Rules: (Root-Left-Right)

- If Root is empty: STOP
- Visit Root
- Execute Preorder on the left sub-tree
- Execute Preorder on the right sub-tree

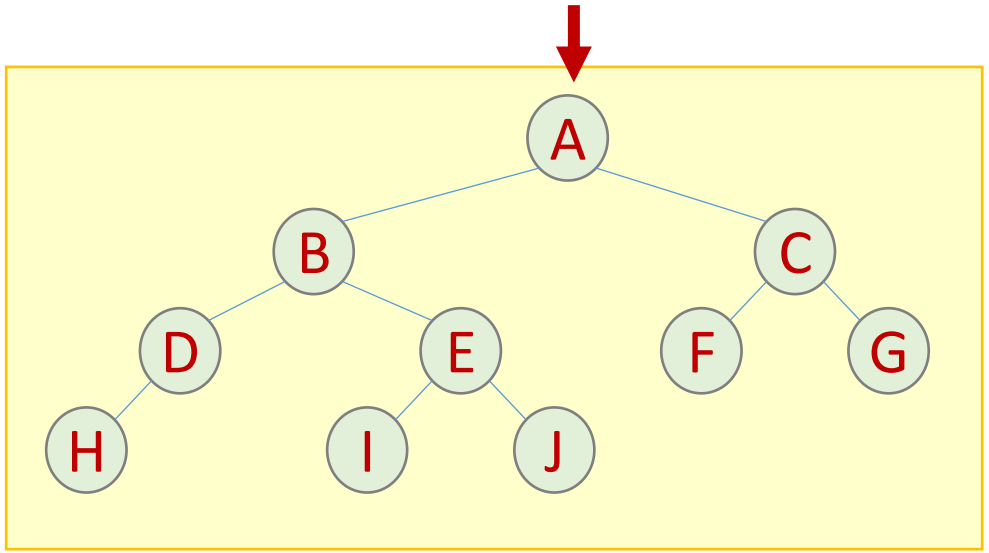


Preorder :

Preorder Traversal

Rules: (Root-Left-Right)

- If Root is empty: STOP
- Visit Root
- Execute Preorder on the left sub-tree
- Execute Preorder on the right sub-tree

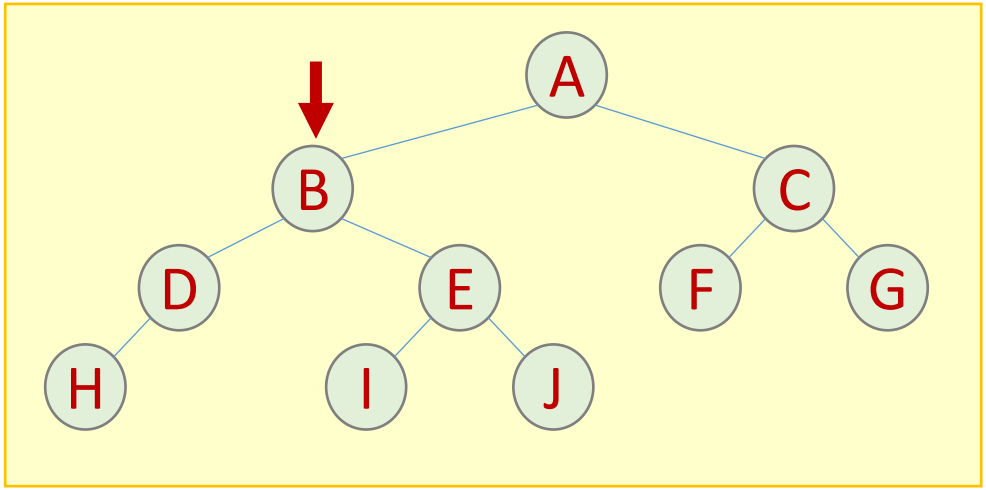


Preorder : A

Preorder Traversal

Rules: (Root-Left-Right)

- If Root is empty: STOP
- Visit Root
- Execute Preorder on the left sub-tree
- Execute Preorder on the right sub-tree

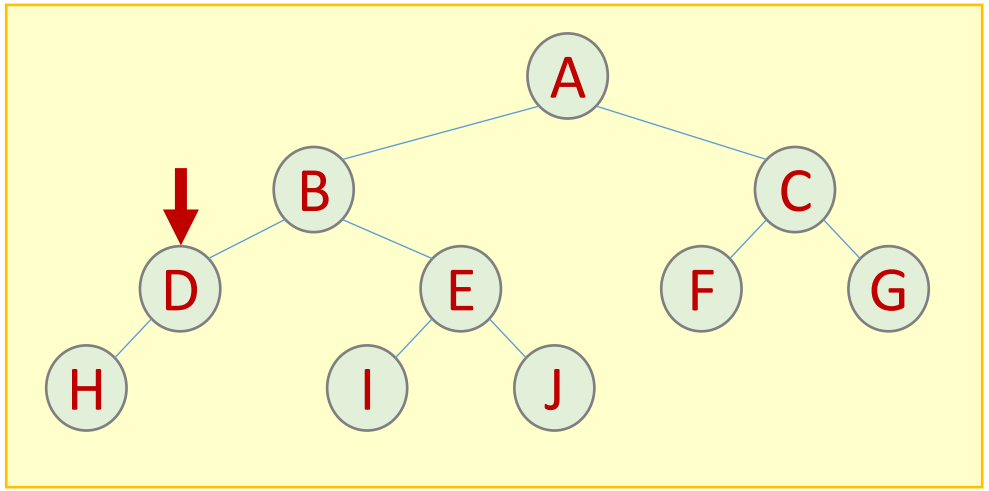


Preorder : A B

Preorder Traversal

Rules: (Root-Left-Right)

- If Root is empty: STOP
- Visit Root
- Execute Preorder on the left sub-tree
- Execute Preorder on the right sub-tree

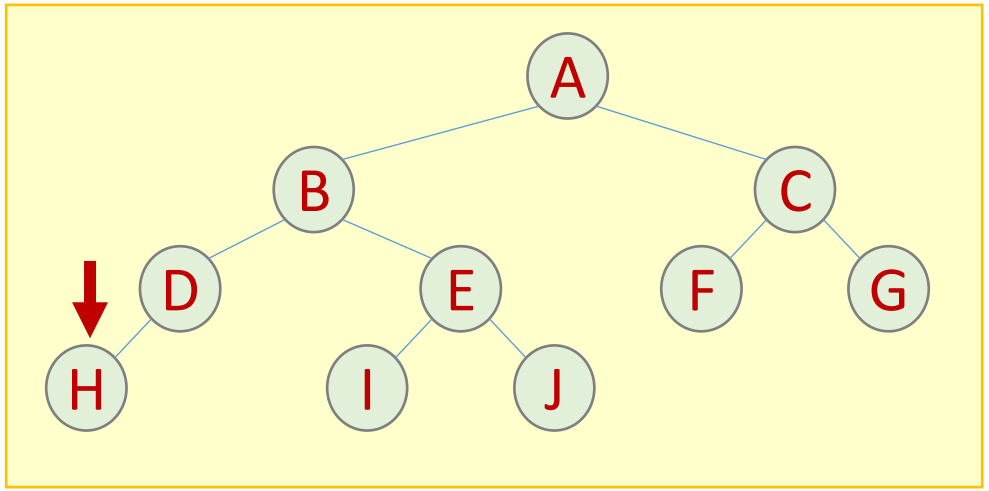


Preorder : A B D

Preorder Traversal

Rules: (Root-Left-Right)

- If Root is empty: STOP
- Visit Root
- Execute Preorder on the left sub-tree
- Execute Preorder on the right sub-tree

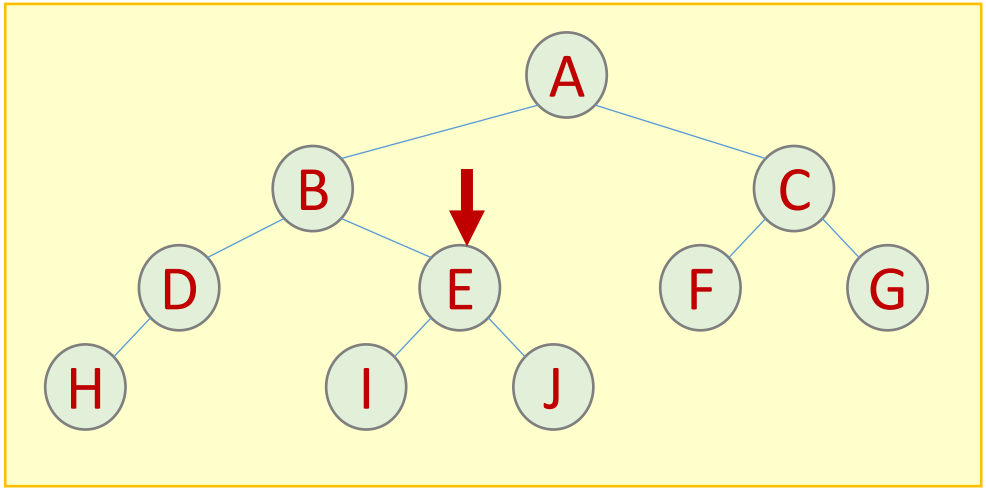


Preorder : A B D H

Preorder Traversal

Rules: (Root-Left-Right)

- If Root is empty: STOP
- Visit Root
- Execute Preorder on the left sub-tree
- Execute Preorder on the right sub-tree

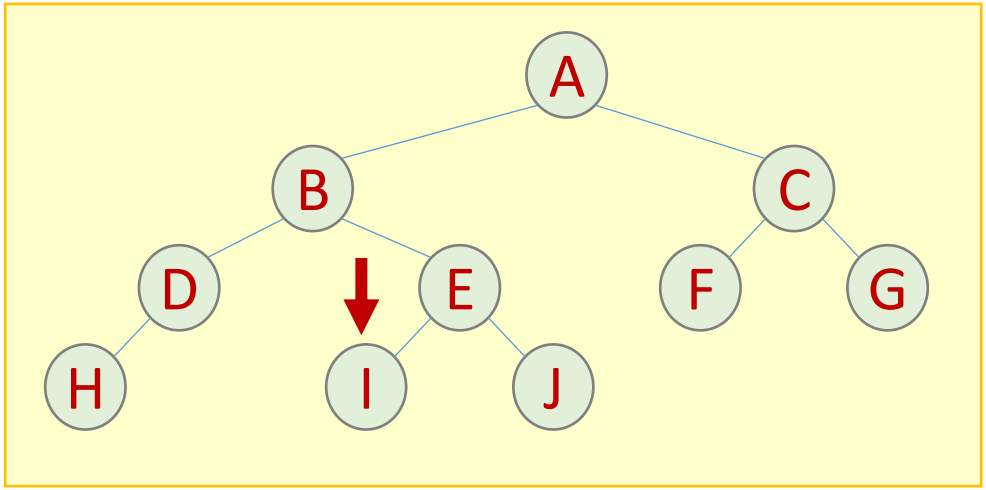


Preorder : A B D H E

Preorder Traversal

Rules: (Root-Left-Right)

- If Root is empty: STOP
- Visit Root
- Execute Preorder on the left sub-tree
- Execute Preorder on the right sub-tree

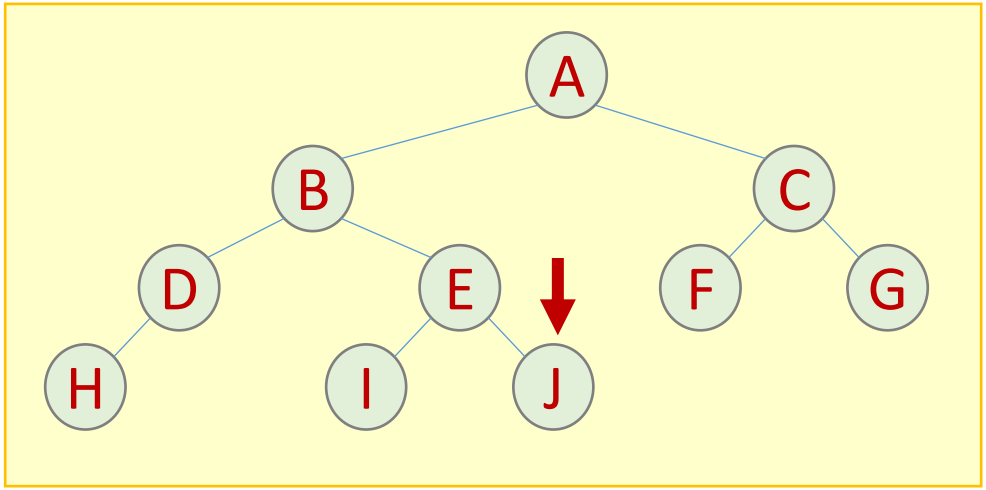


Preorder : A B D H E I

Preorder Traversal

Rules: (Root-Left-Right)

- If Root is empty: STOP
- Visit Root
- Execute Preorder on the left sub-tree
- Execute Preorder on the right sub-tree

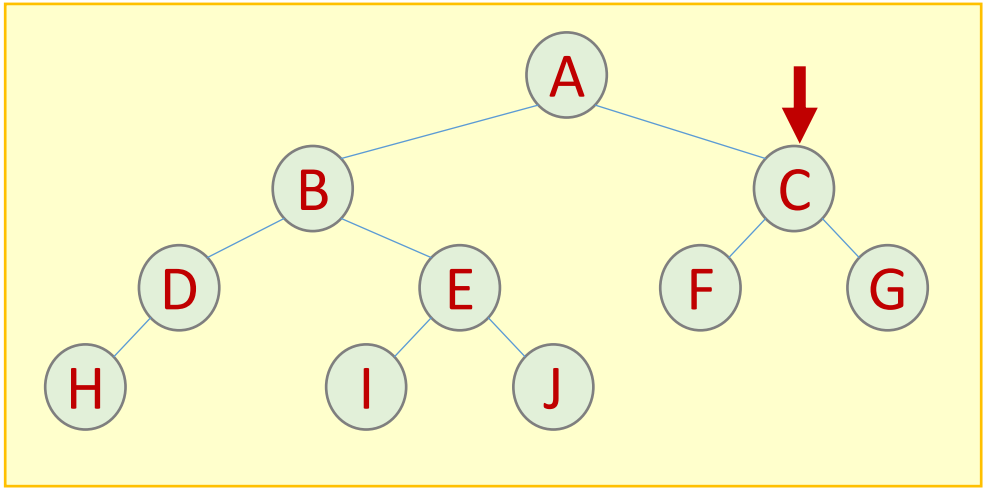


Preorder : A B D H E I J

Preorder Traversal

Rules: (Root-Left-Right)

- If Root is empty: STOP
- Visit Root
- Execute Preorder on the left sub-tree
- Execute Preorder on the right sub-tree

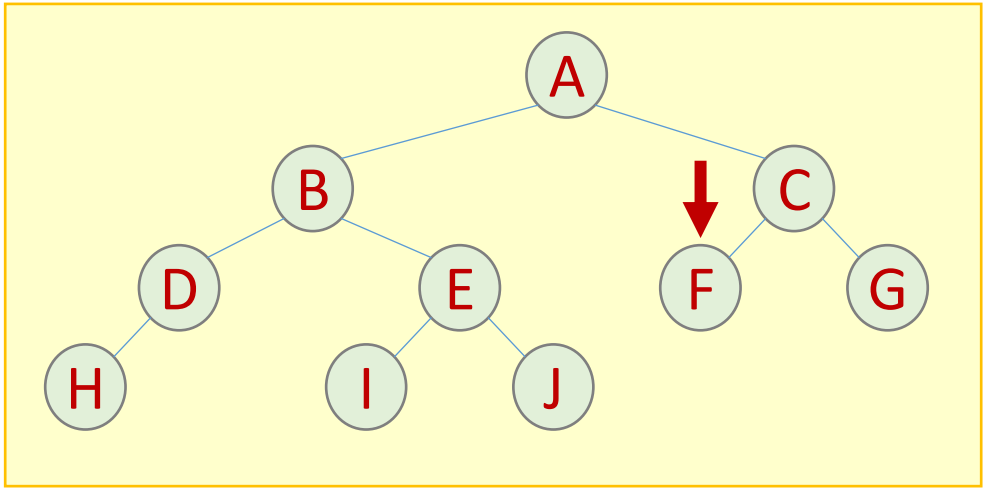


Preorder : A B D H E I J C

Preorder Traversal

Rules: (Root-Left-Right)

- If Root is empty: STOP
- Visit Root
- Execute Preorder on the left sub-tree
- Execute Preorder on the right sub-tree

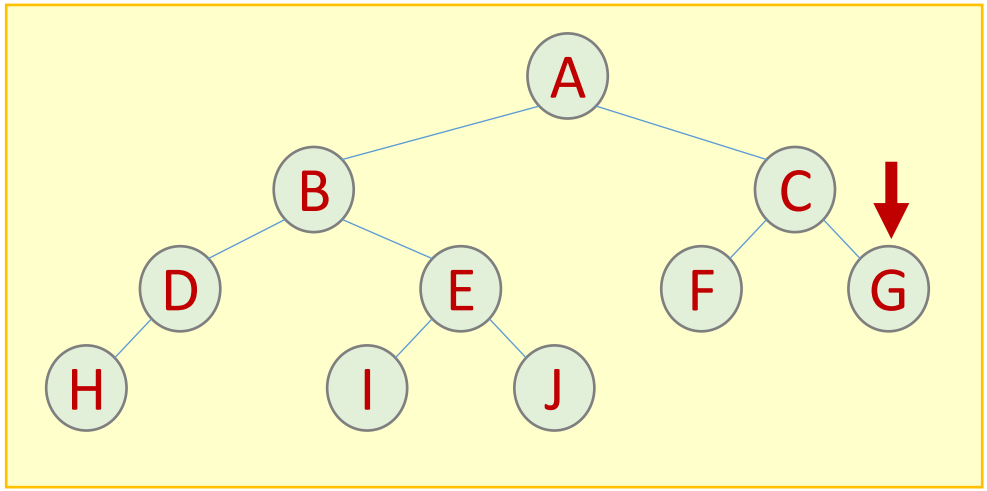


Preorder : A B D H E I J C F

Preorder Traversal

Rules: (Root-Left-Right)

- If Root is empty: STOP
- Visit Root
- Execute Preorder on the left sub-tree
- Execute Preorder on the right sub-tree



Preorder : A B D H E I J C F G

Preorder Traversal

Algorithm preOrder (val root <node pointer>)

Traverse a tree in node-left-right (preorder) sequence.

Pre: root is the entry node of a tree or subtree

Post: each node has been processed inorder

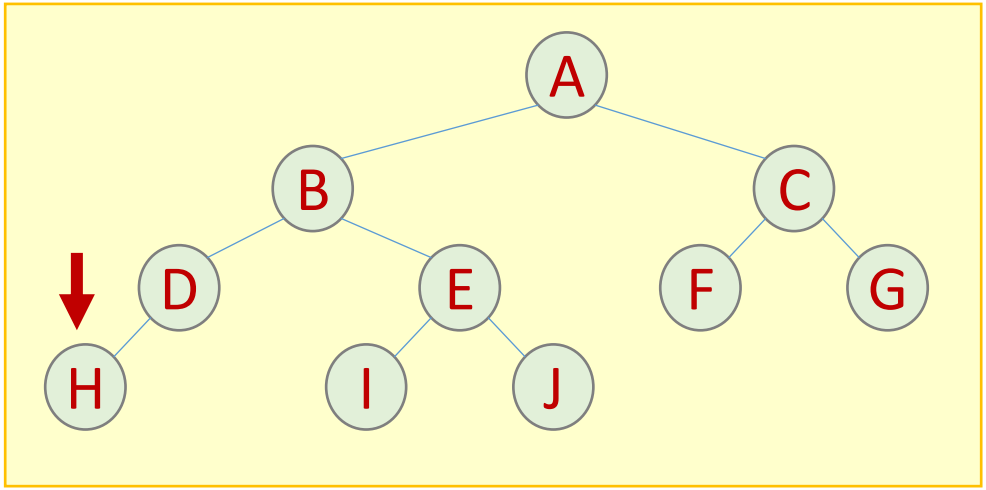
1. **If (root is not null)**
    1. **Process (root)**
    2. **preOrder (root.leftSubtree)**
    3. **preOrder (root.rightSubtree)**
2. **End if**
3. **Return**

End preOrder

Inorder Traversal

Rules: (Left-Root-Right)

- If Root is empty: STOP
- Execute Inorder on the left sub-tree
- Visit Root
- Execute Inorder on the right sub-tree

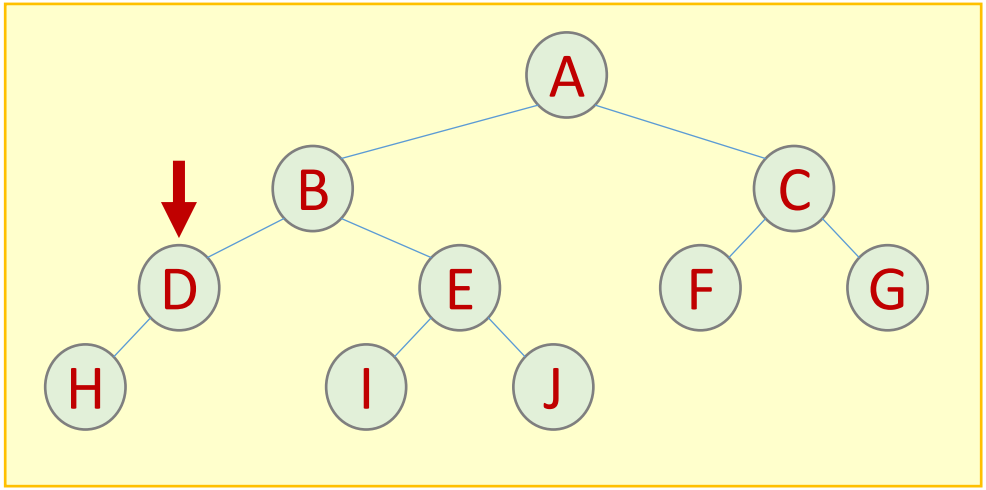


Inorder : H

Inorder Traversal

Rules: (Left-Root-Right)

- If Root is empty: STOP
- Execute Inorder on the left sub-tree
- Visit Root
- Execute Inorder on the right sub-tree

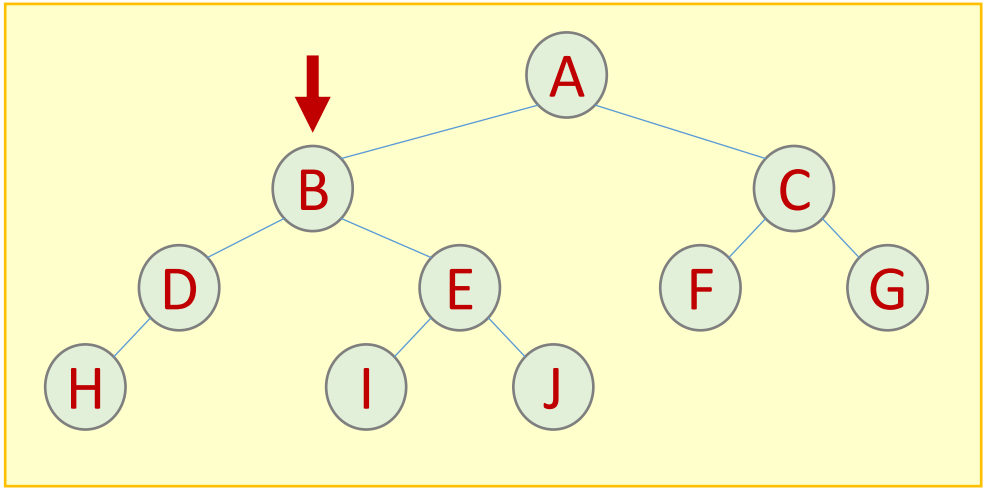


Inorder : H D

Inorder Traversal

Rules: (Left-Root-Right)

- If Root is empty: STOP
- Execute Inorder on the left sub-tree
- Visit Root
- Execute Inorder on the right sub-tree

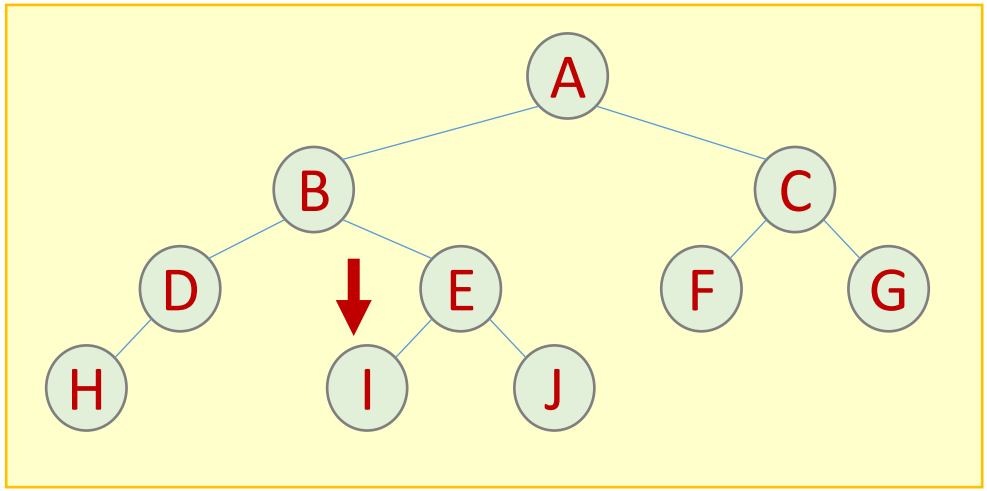


Inorder : H D B

Inorder Traversal

Rules: (Left-Root-Right)

- If Root is empty: STOP
- Execute Inorder on the left sub-tree
- Visit Root
- Execute Inorder on the right sub-tree

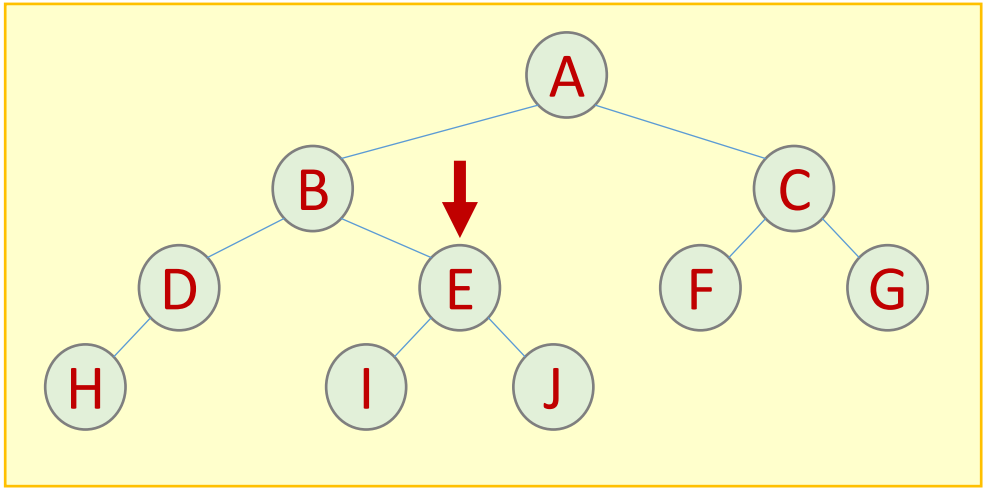


Inorder : H D B I

Inorder Traversal

Rules: (Left-Root-Right)

- If Root is empty: STOP
- Execute Inorder on the left sub-tree
- Visit Root
- Execute Inorder on the right sub-tree

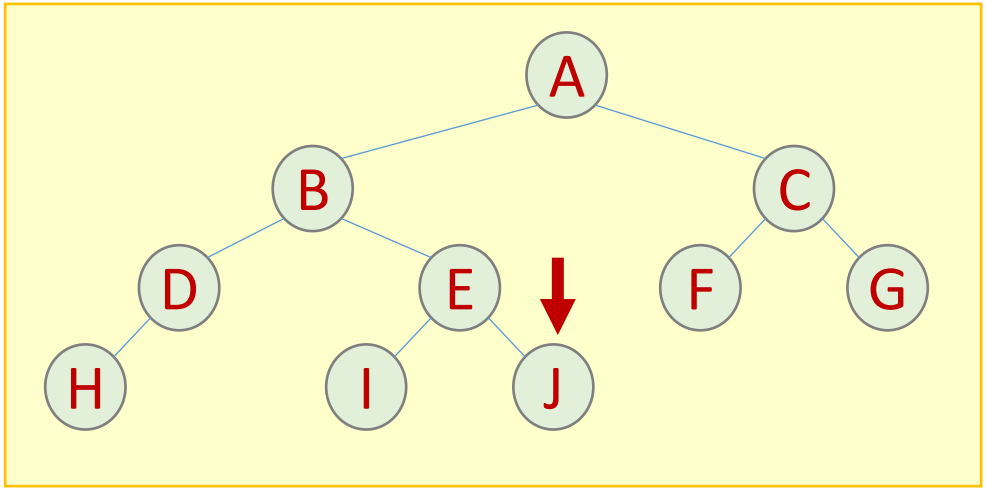


Inorder : H D B I E

Inorder Traversal

Rules: (Left-Root-Right)

- If Root is empty: STOP
- Execute Inorder on the left sub-tree
- Visit Root
- Execute Inorder on the right sub-tree

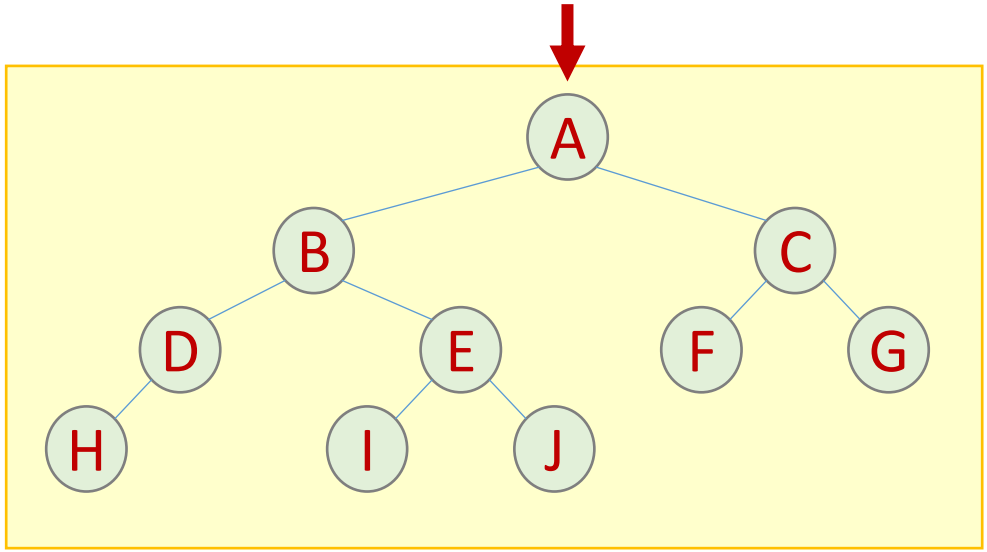


Inorder : H D B I E J

Inorder Traversal

Rules: (Left-Root-Right)

- If Root is empty: STOP
- Execute Inorder on the left sub-tree
- Visit Root
- Execute Inorder on the right sub-tree

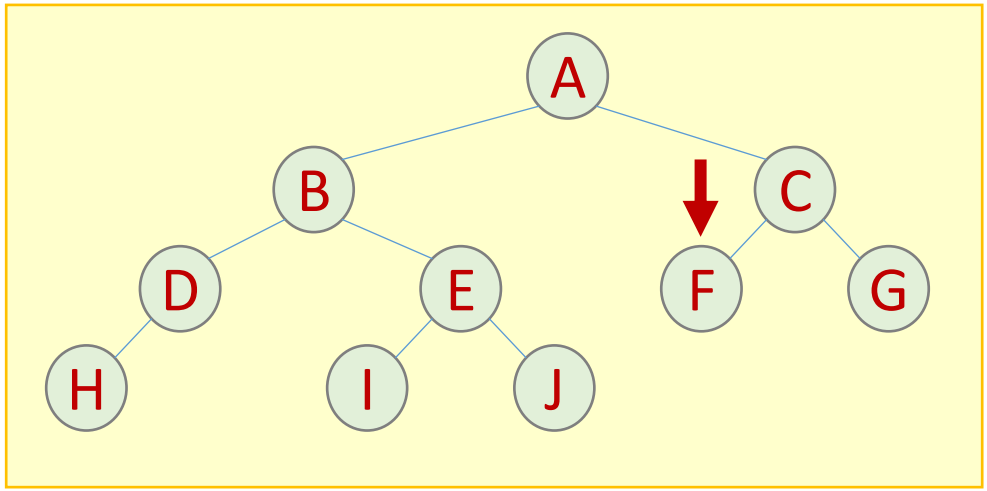


Inorder : H D B I E J A

Inorder Traversal

Rules: (Left-Root-Right)

- If Root is empty: STOP
- Execute Inorder on the left sub-tree
- Visit Root
- Execute Inorder on the right sub-tree

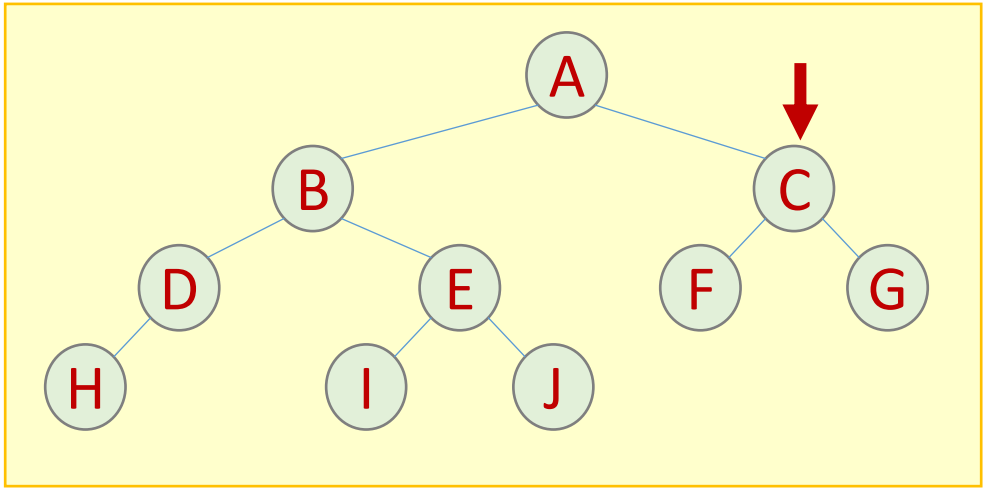


Inorder : H D B I E J A F

Inorder Traversal

Rules: (Left-Root-Right)

- If Root is empty: STOP
- Execute Inorder on the left sub-tree
- Visit Root
- Execute Inorder on the right sub-tree

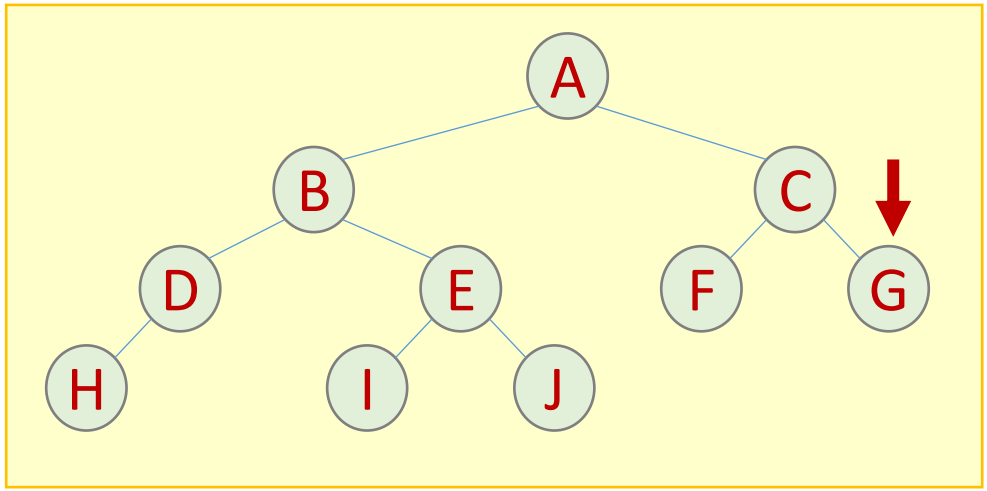


Inorder : H D B I E J A F C

Inorder Traversal

Rules: (Left-Root-Right)

- If Root is empty: STOP
- Execute Inorder on the left sub-tree
- Visit Root
- Execute Inorder on the right sub-tree



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

Inorder Traversal

Algorithm inOrder (val root <node pointer>)

Traverse a tree in left-node-right (inorder) sequence.

Pre: root is the entry node of a tree or subtree

Post: each node has been processed in order

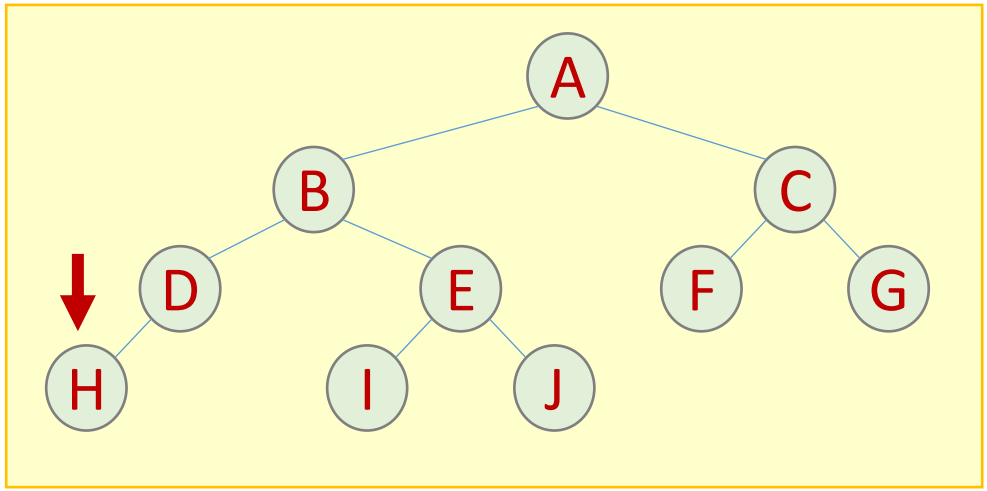
1. **If (root is not null)**
    1. **inOrder (root.leftSubTree)**
    2. **Process (root)**
    3. **inOrder (root.rightSubTree)**
2. **End if**
3. **Return**

End inOrder

Postorder Traversal

Rules: (Left-Right-Root)

- If Root is empty: STOP
- Execute Postorder on the left sub-tree
- Execute Postorder on the right sub-tree
- Visit Root

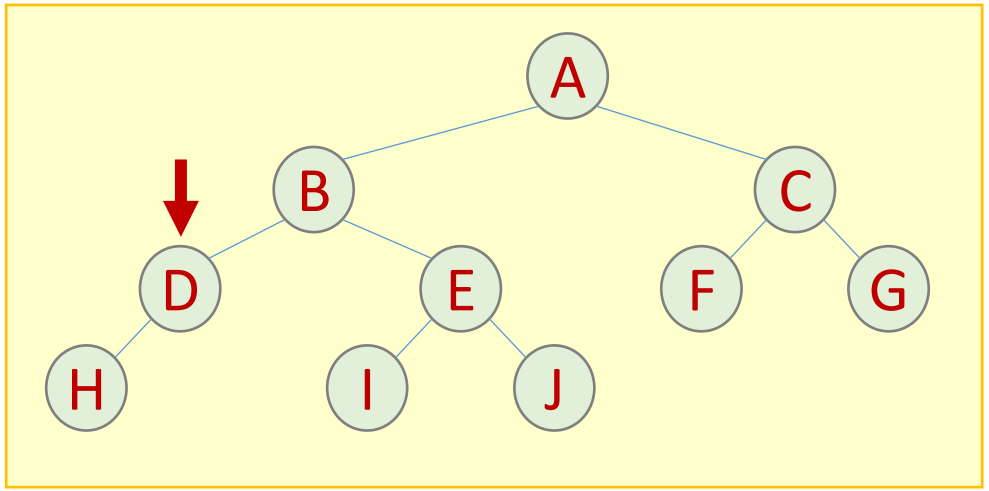


Postorder : H

Postorder Traversal

Rules: (Left-Right-Root)

- If Root is empty: STOP
- Execute Postorder on the left sub-tree
- Execute Postorder on the right sub-tree
- Visit Root

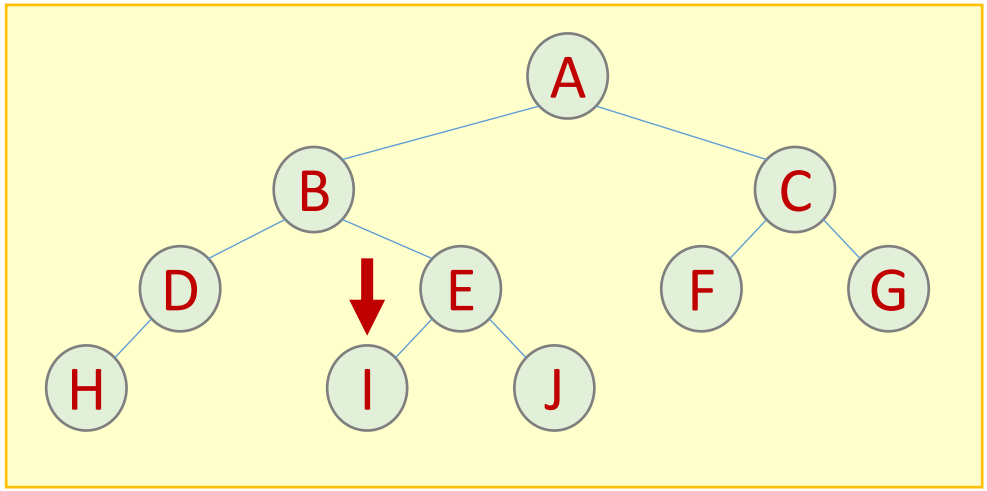


Postorder : H D

Postorder Traversal

Rules: (Left-Right-Root)

- If Root is empty: STOP
- Execute Postorder on the left sub-tree
- Execute Postorder on the right sub-tree
- Visit Root

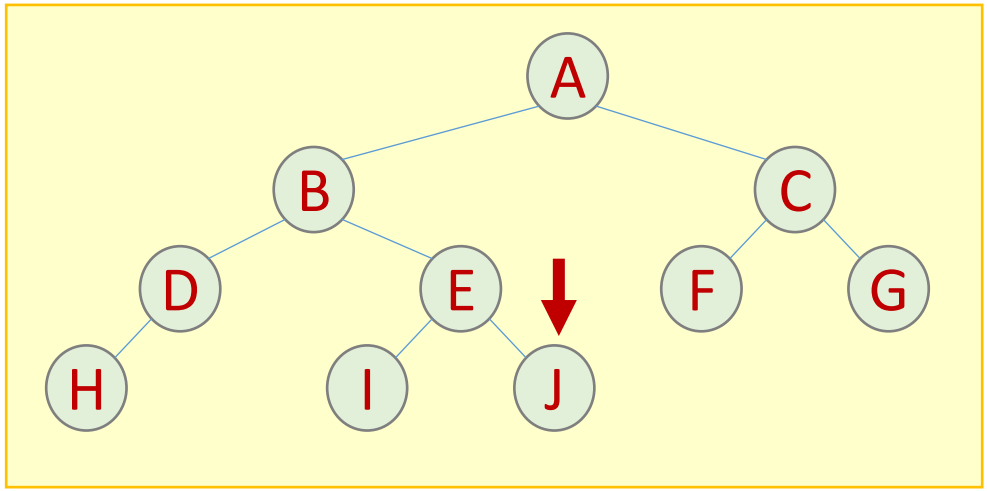


Postorder : H D I

Postorder Traversal

Rules: (Left-Right-Root)

- If Root is empty: STOP
- Execute Postorder on the left sub-tree
- Execute Postorder on the right sub-tree
- Visit Root

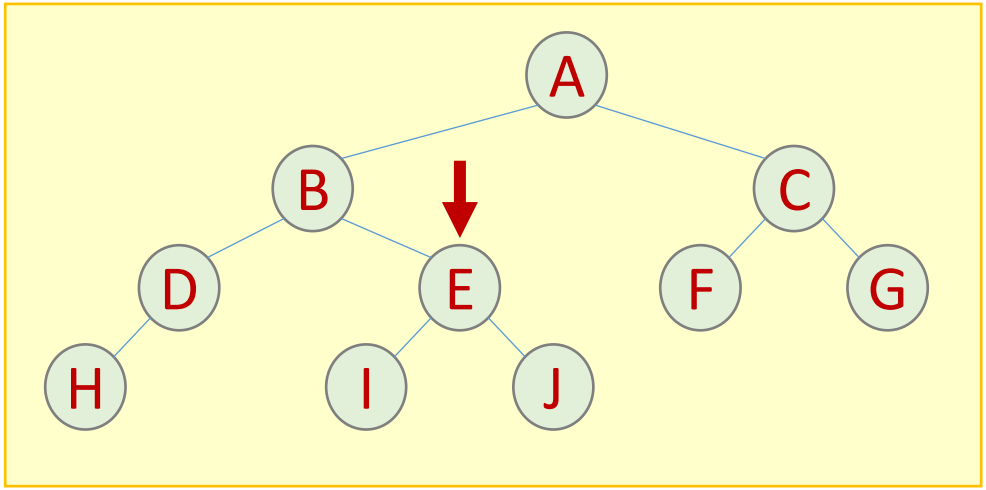


Postorder : H D I J

Postorder Traversal

Rules: (Left-Right-Root)

- If Root is empty: STOP
- Execute Postorder on the left sub-tree
- Execute Postorder on the right sub-tree
- Visit Root

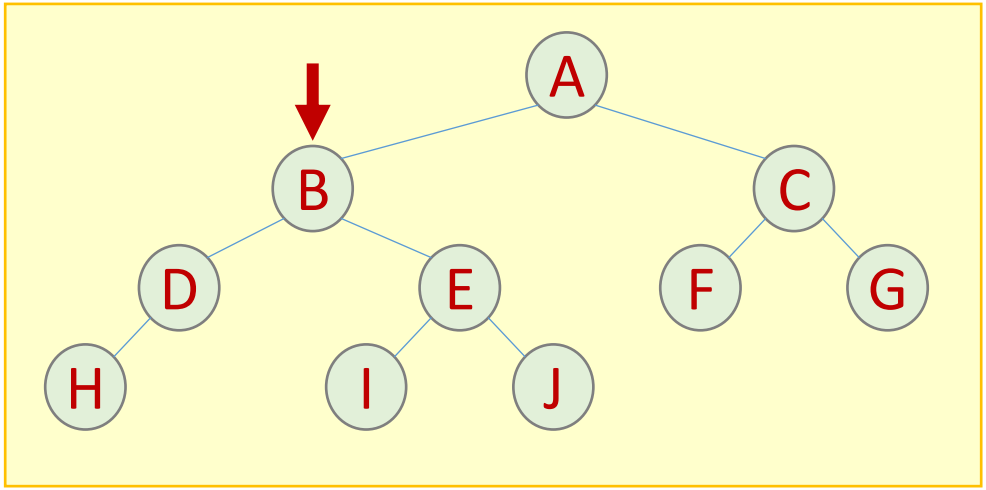


Postorder : H D I J E

Postorder Traversal

Rules: (Left-Right-Root)

- If Root is empty: STOP
- Execute Postorder on the left sub-tree
- Execute Postorder on the right sub-tree
- Visit Root

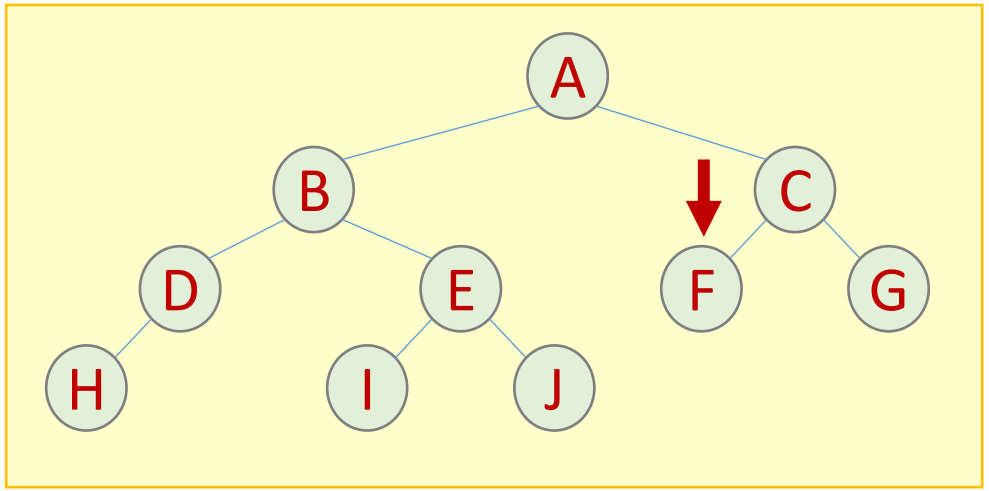


Postorder : H D I J E B

Postorder Traversal

Rules: (Left-Right-Root)

- If Root is empty: STOP
- Execute Postorder on the left sub-tree
- Execute Postorder on the right sub-tree
- Visit Root

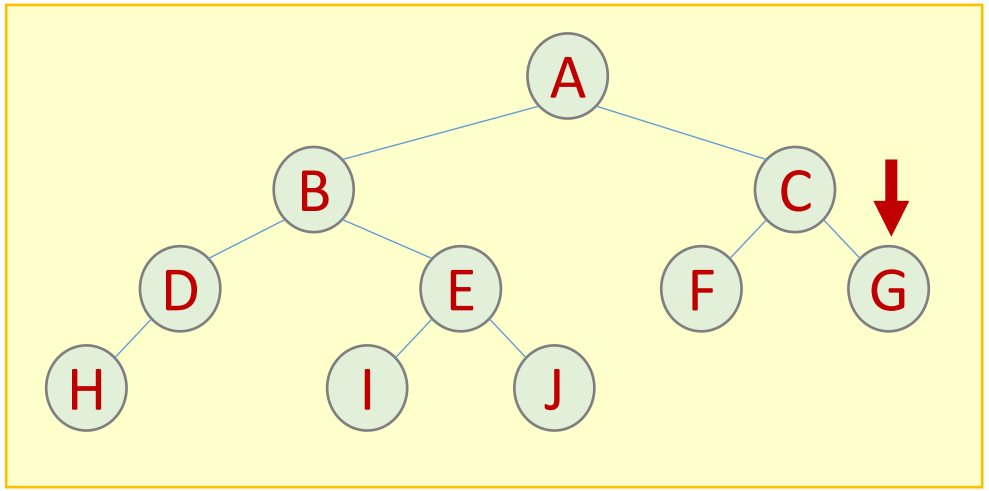


Postorder : H D I J E B F

Postorder Traversal

Rules: (Left-Right-Root)

- If Root is empty: STOP
- Execute Postorder on the left sub-tree
- Execute Postorder on the right sub-tree
- Visit Root

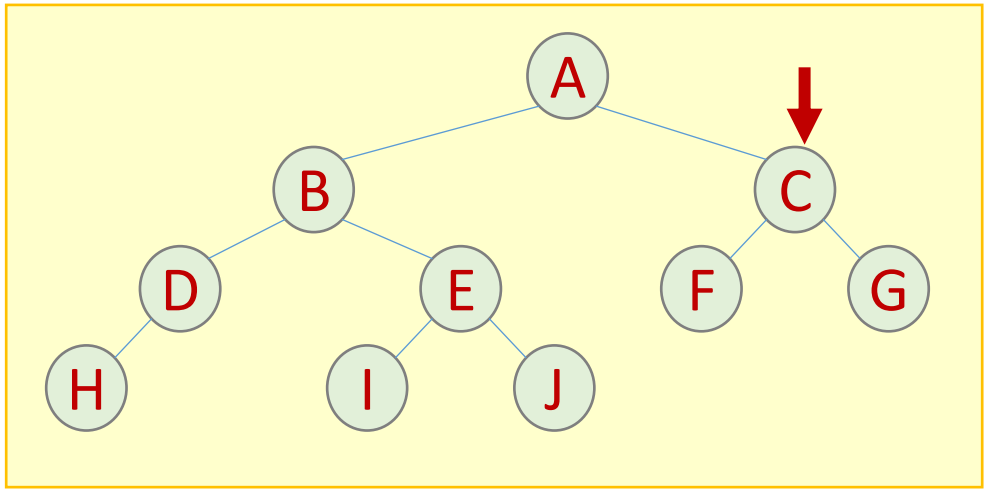


Postorder : H D I J E B F G

Postorder Traversal

Rules: (Left-Right-Root)

- If Root is empty: STOP
- Execute Postorder on the left sub-tree
- Execute Postorder on the right sub-tree
- Visit Root

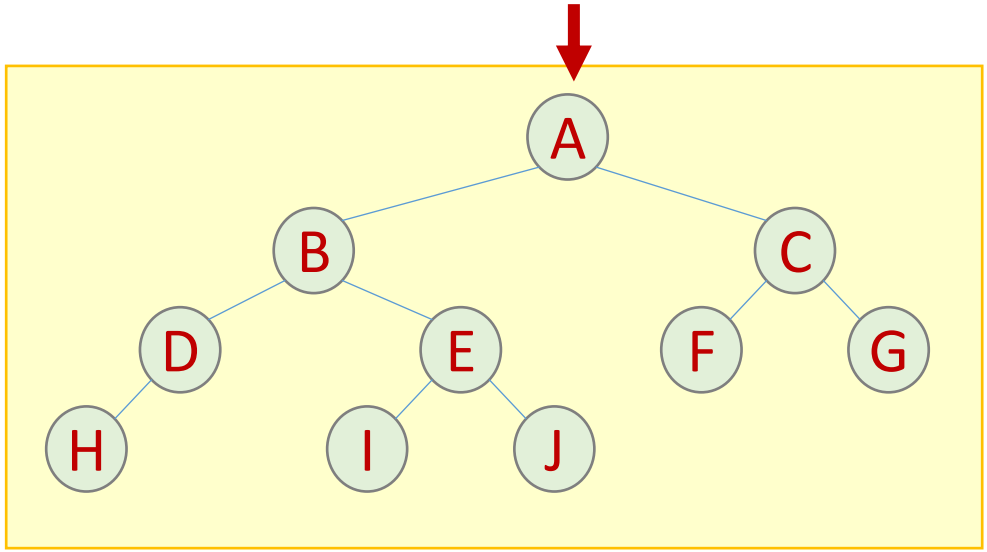


Postorder : H D I J E B F G C

Postorder Traversal

Rules: (Left-Right-Root)

- If Root is empty: STOP
- Execute Postorder on the left sub-tree
- Execute Postorder on the right sub-tree
- Visit Root



Postorder : H D I J E B F G C A

Postorder Traversal

Algorithm postOrder (val root <node pointer>)

Traverse a tree in left-right-node (postorder) sequence.

Pre: root is the entry node of a tree or subtree

Post: each node has been processed in order

1. **If (root is not null)**
    1. **postOrder (root.leftSubtree)**
    2. **postOrder (root.rightSubtree)**
    3. **Process (root)**
2. **End if**
3. **Return**

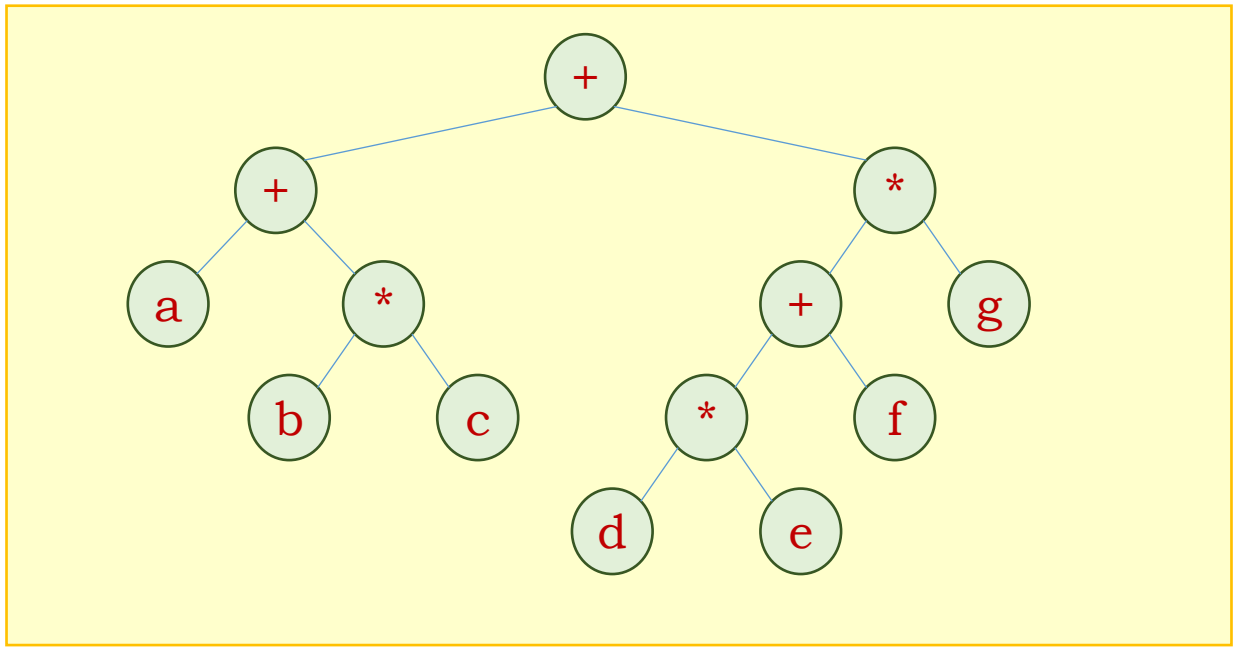
End postOrder

Binary Tree - Application

- One of the most interesting application of binary tree application is the expression tree.
- An expression tree is a binary tree with the following properties:
  - Each leaf is an operand
  - The root and internal nodes are operators
  - Subtrees are subexpressions, with the root being an operator.

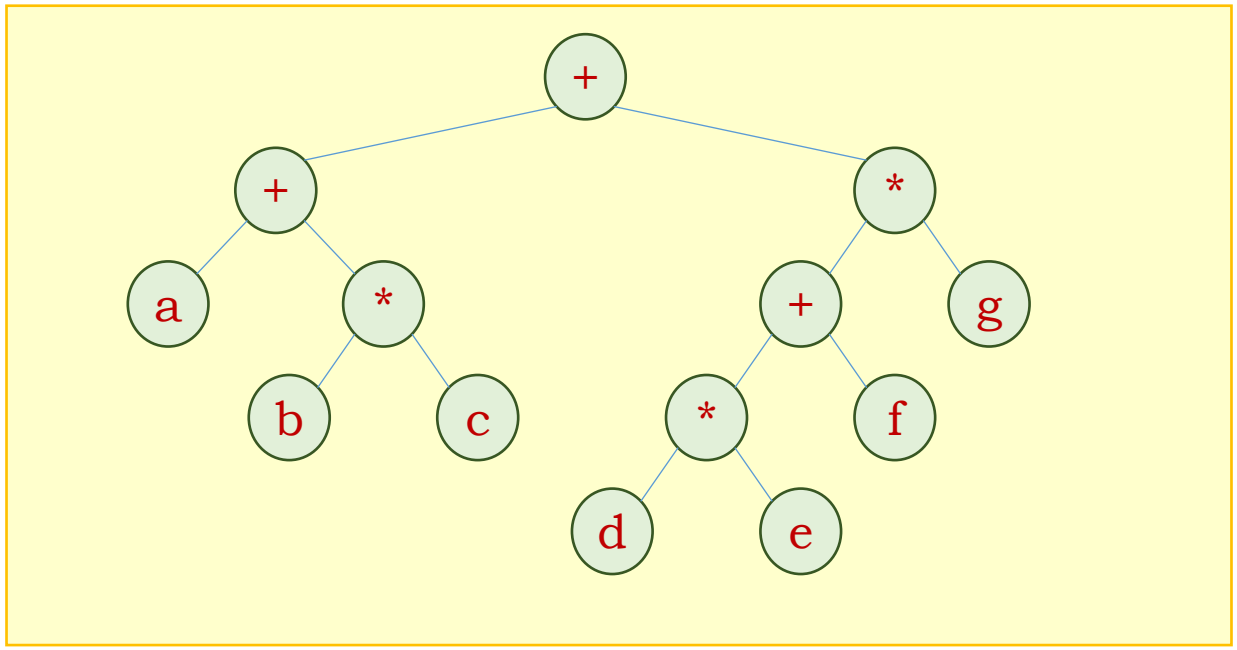
Binary Tree – Expression Tree

An Example: Expression Trees



Binary Tree – Expression Tree

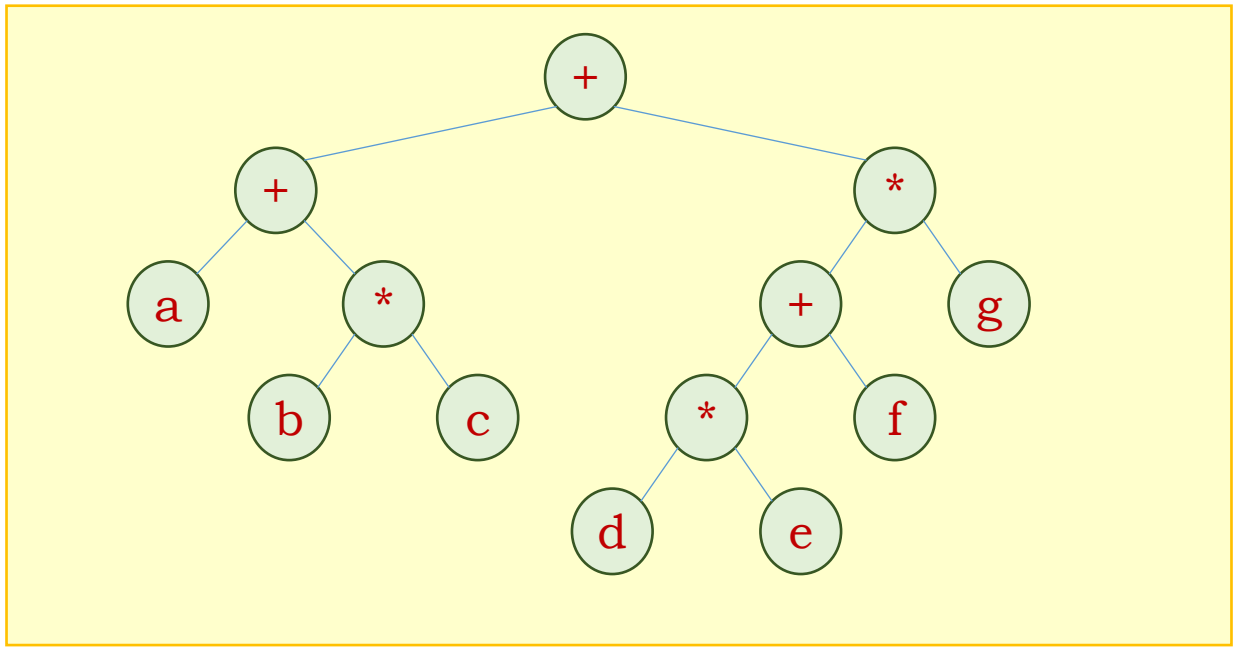
An Example: Expression Trees



Preorder: **+ + a * b c * + * d e f g**

Binary Tree – Expression Tree

An Example: Expression Trees

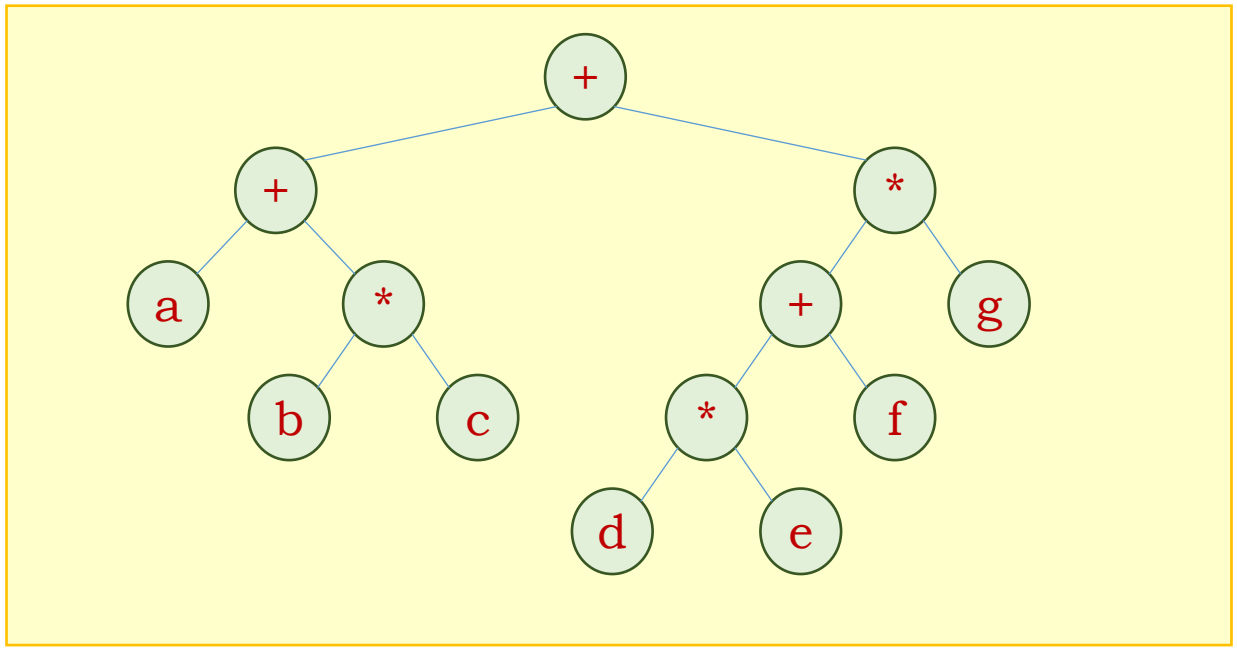


Preorder: **+ + a * b c * + * d e f g**

Inorder: **a + b * c + d * e + f * g**

Binary Tree – Expression Tree

An Example: Expression Trees



Preorder: **+ + a * b c * + * d e f g**

Inorder: **a + b * c + d * e + f * g**

Postorder: **a b c * + d e * f + g * +**

Binary Tree – Expression Tree

Algorithm prefix (val tree <tree pointer>)

Print the prefix expression for an expression tree.

Pre: tree is a pointer to an expression tree

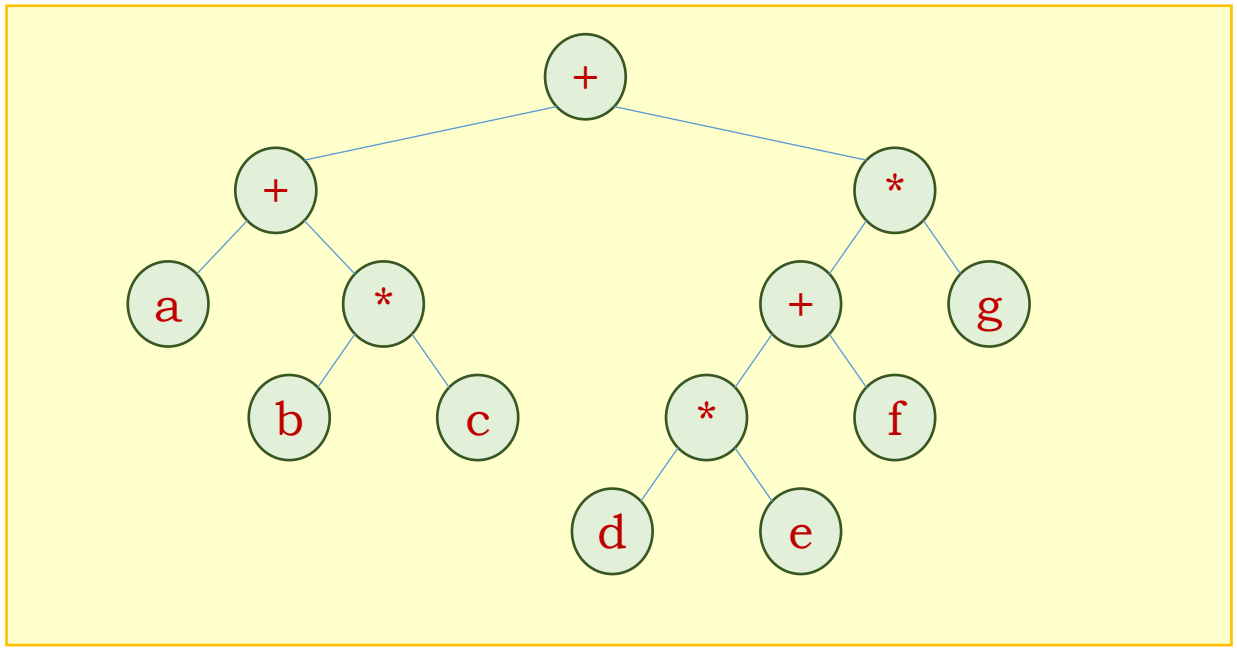
Post: the prefix expression has been printed

1. **If (tree not empty)**
    1. **Print (tree.token)**
    2. **Prefix (tree.left)**
    3. **Prefix (tree.right)**
2. **End if**
3. **Return**

End prefix

Binary Tree – Expression Tree

An Example: Expression Trees



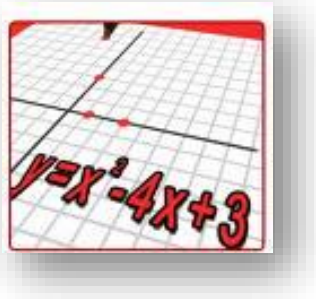
Preorder: **+ + a * b c * + * d e f g**

Binary Tree – Expression Tree

Algorithm infix (val tree <tree pointer>)
Print the infix expression for an expression tree.
Pre: Tree is a pointer to an expression tree
Post: the infix expression has been printed

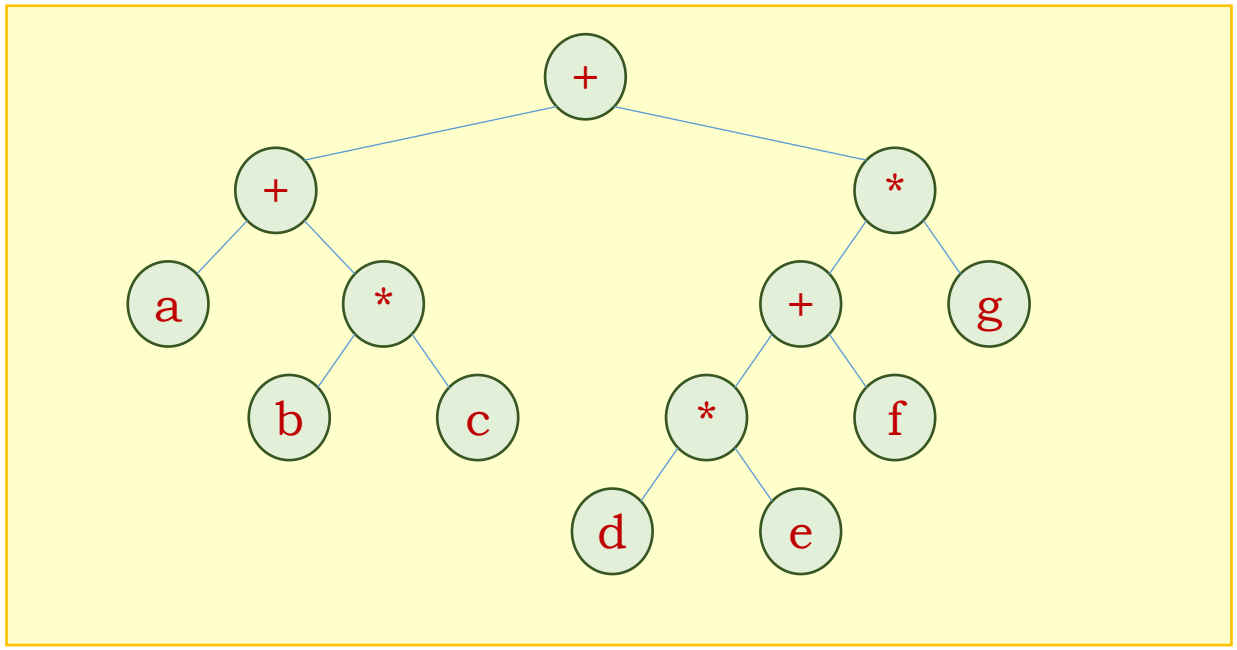
1. **If (tree not empty)**
    1. **If (tree.token is an operand)**
        1. **Print (tree.token)**
    2. **Else**
        1. **Print (open parenthesis)**
        2. **Infix (tree.left)**
        3. **Print (tree.token)**
        4. **Infix (tree.right)**
        5. **Print (close parenthesis)**
    3. **End if**
2. **End if**
3. **Return**

End infix



Binary Tree – Expression Tree

An Example: Expression Trees



Inorder: **((a + (b * c)) + (((d * e) + f) * g))**

Binary Tree – Expression Tree

Algorithm postfix (val tree <tree pointer>)

Print the postfix expression for an expression tree.

Pre: tree is a pointer to an expression tree

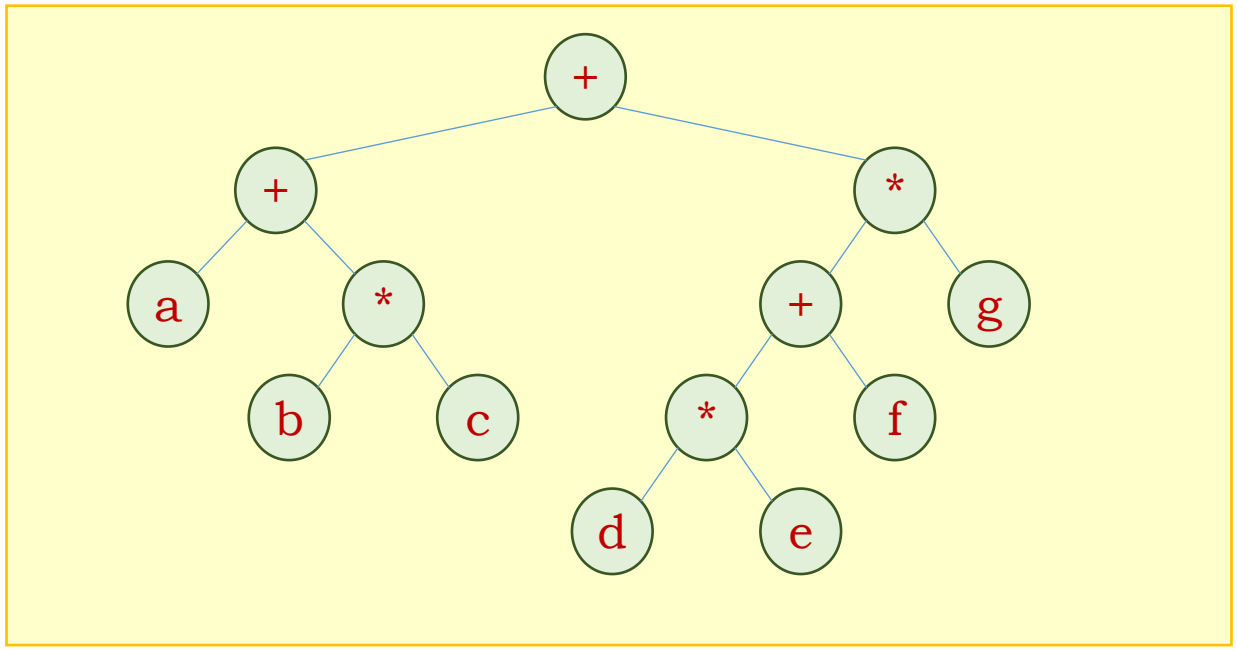
Post: the postfix expression has been printed

1. **If (tree not empty)**
    1. **Postfix (tree.left)**
    2. **Postfix (tree.right)**
    3. **Print (tree.token)**
2. **End if**
3. **Return**

End postfix

Binary Tree – Expression Tree

An Example: Expression Trees



Postorder: **a b c * + d e * f + g * +**

Postfix expression and Stack

- A stack can be useful when the application requires that the use of data be postponed for a while.
- We see how we can use stack postponement to construct the expression tree.

Postfix expression and Stack

For example, given the postfix expression, we want to construct the expression tree.

```
a b + c d + e * *
```

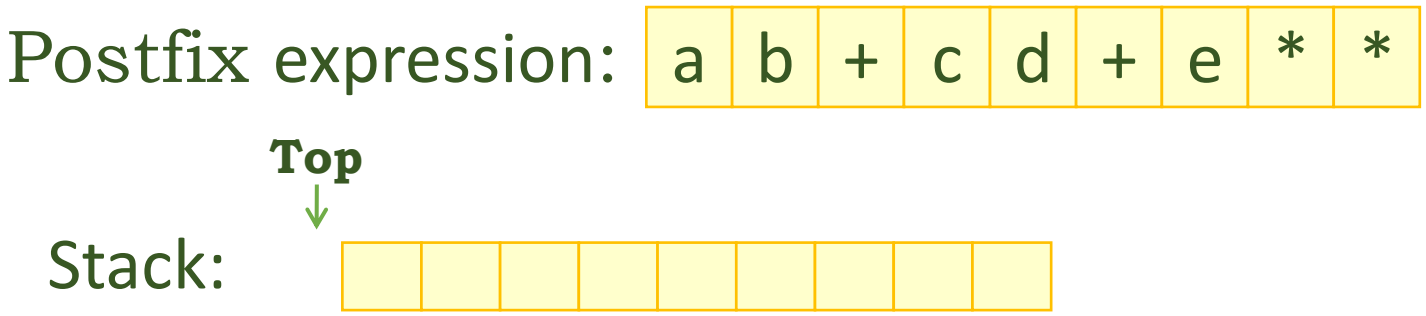
- We will use a stack of references to keep the references to the nodes of the expression tree.

Postfix expression and Stack

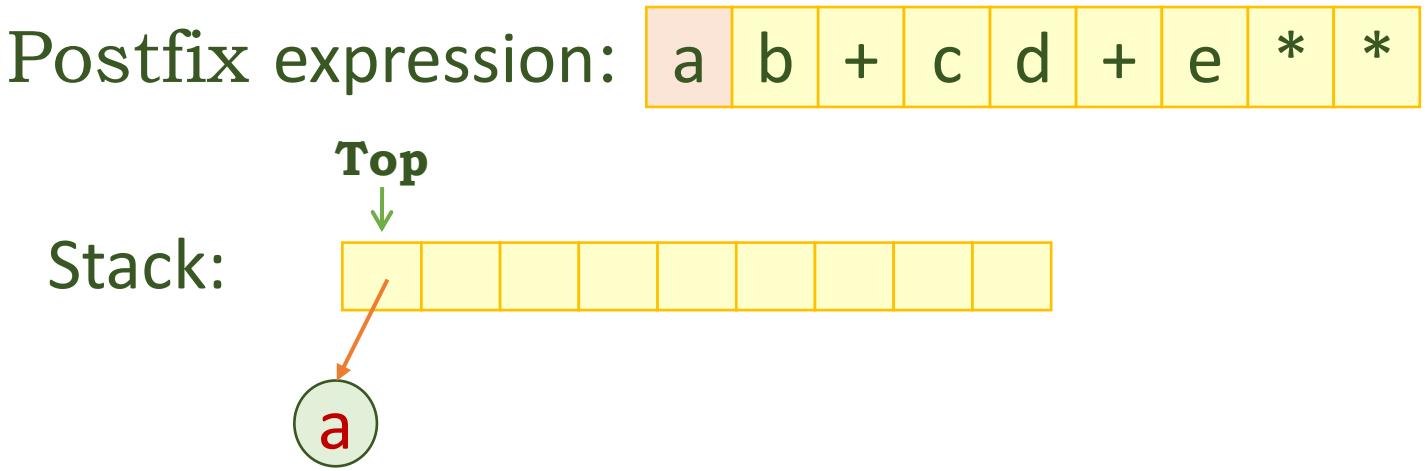
Method:

- Evaluate the expression from left to right:
  - When we find an operand, we dynamically create a node and set the operand as the content of the node. The reference to the node is pushed to the stack.
  - When we find an operator, we dynamically create a node and set the operator as the content of the node. We refer to this node as the current node.
    - We then pop the two references to the operands at the top of the stack.
    - The first reference is set to be the right subtree of the current node.
    - The second reference is set to the left subtree of the current node.
  - We push the reference to the current node to the stack.

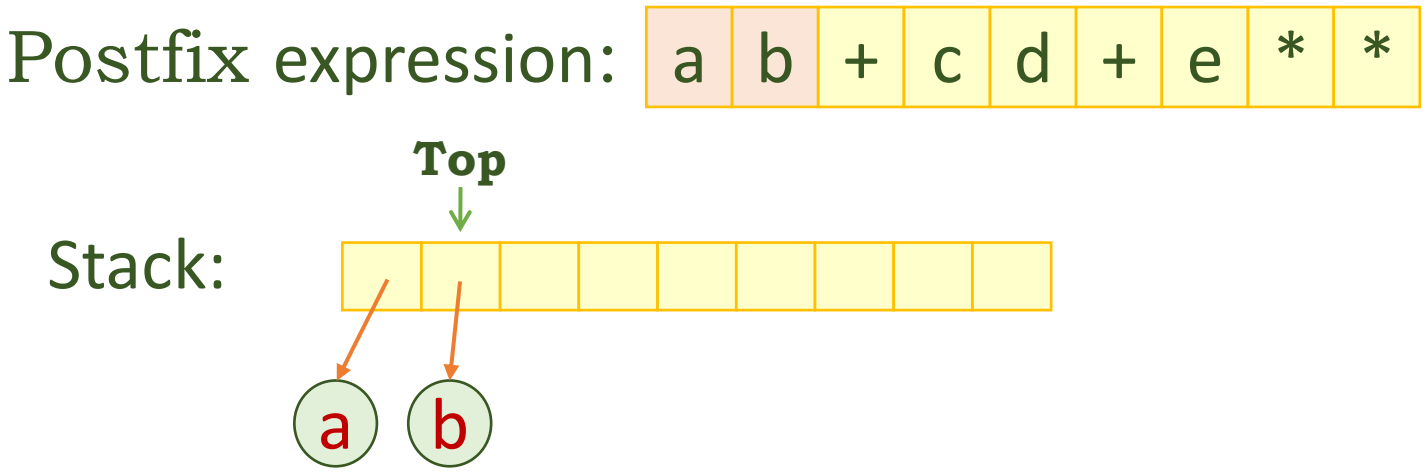
Postfix expression and Stack



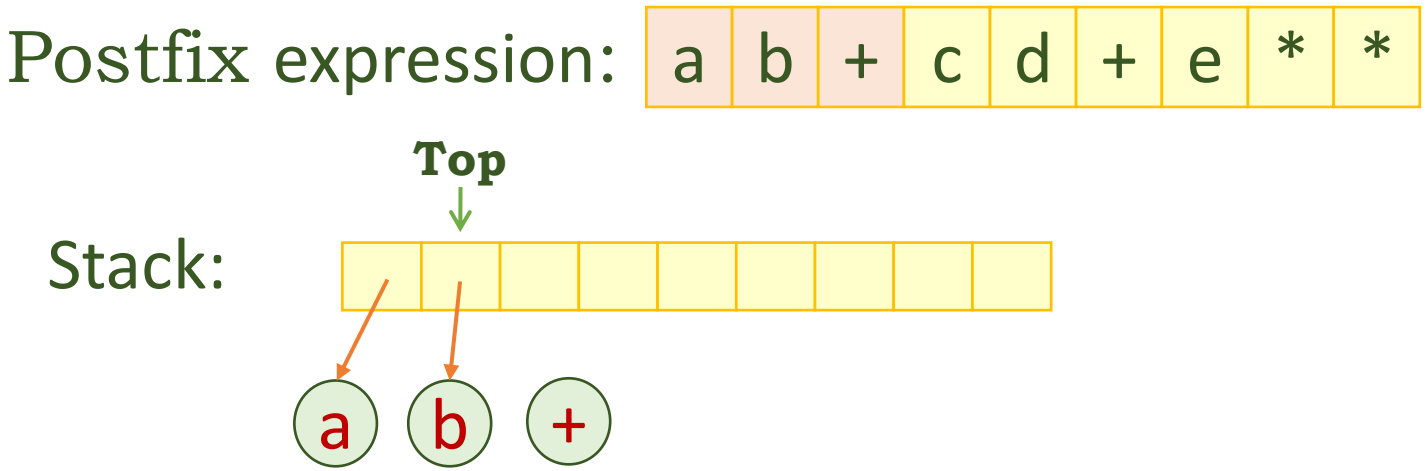
Postfix expression and Stack



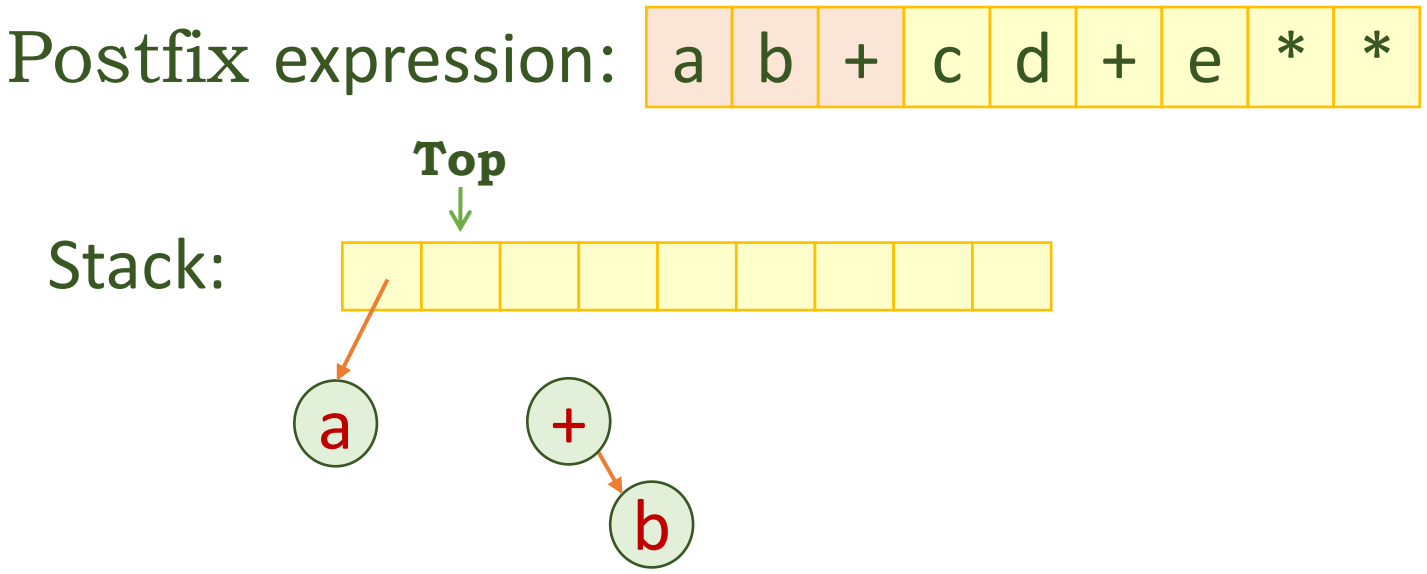
Postfix expression and Stack



Postfix expression and Stack



Postfix expression and Stack

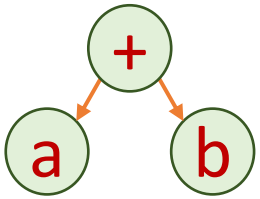


Postfix expression and Stack

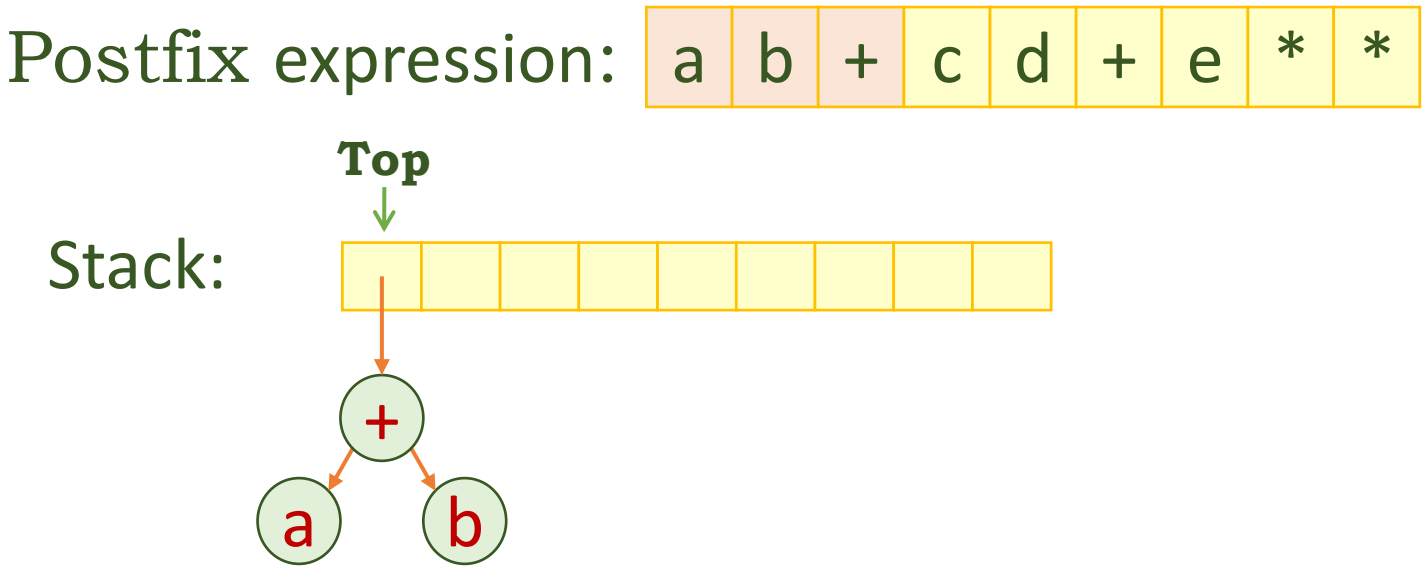
Postfix expression:

a	b	+	c	d	+	e	*	*

Stack:



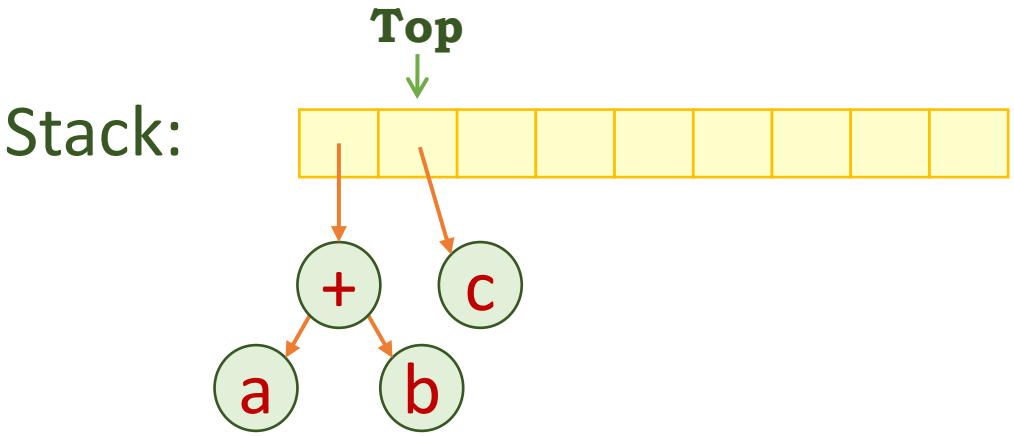
Postfix expression and Stack



Postfix expression and Stack

Postfix expression:

a	b	+	c	d	+	e	*	*

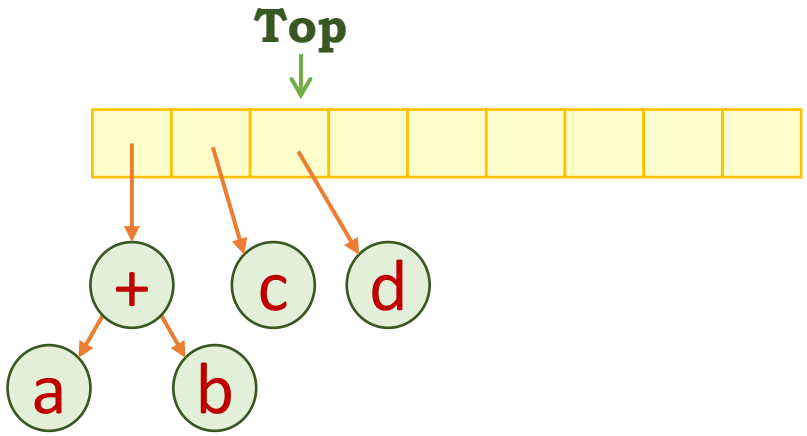


Postfix expression and Stack

Postfix expression:

a	b	+	c	d	+	e	*	*

Stack:

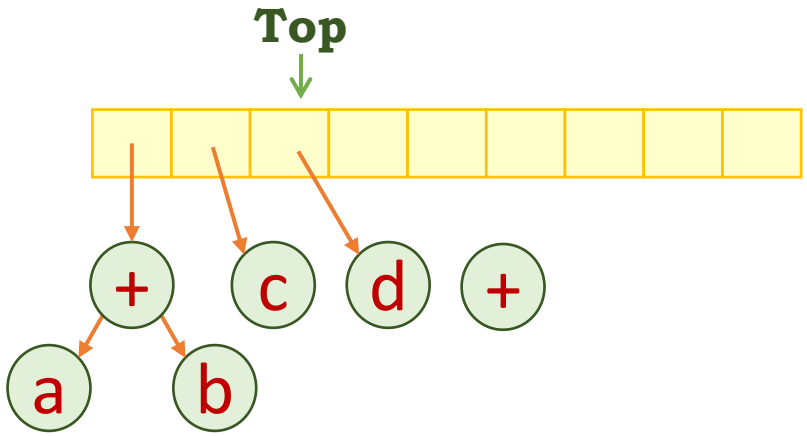


Postfix expression and Stack

Postfix expression:

a	b	+	c	d	+	e	*	*

Stack:

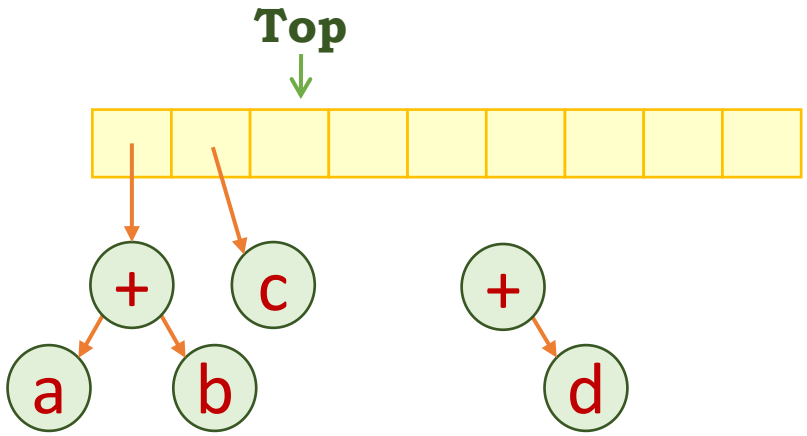


Postfix expression and Stack

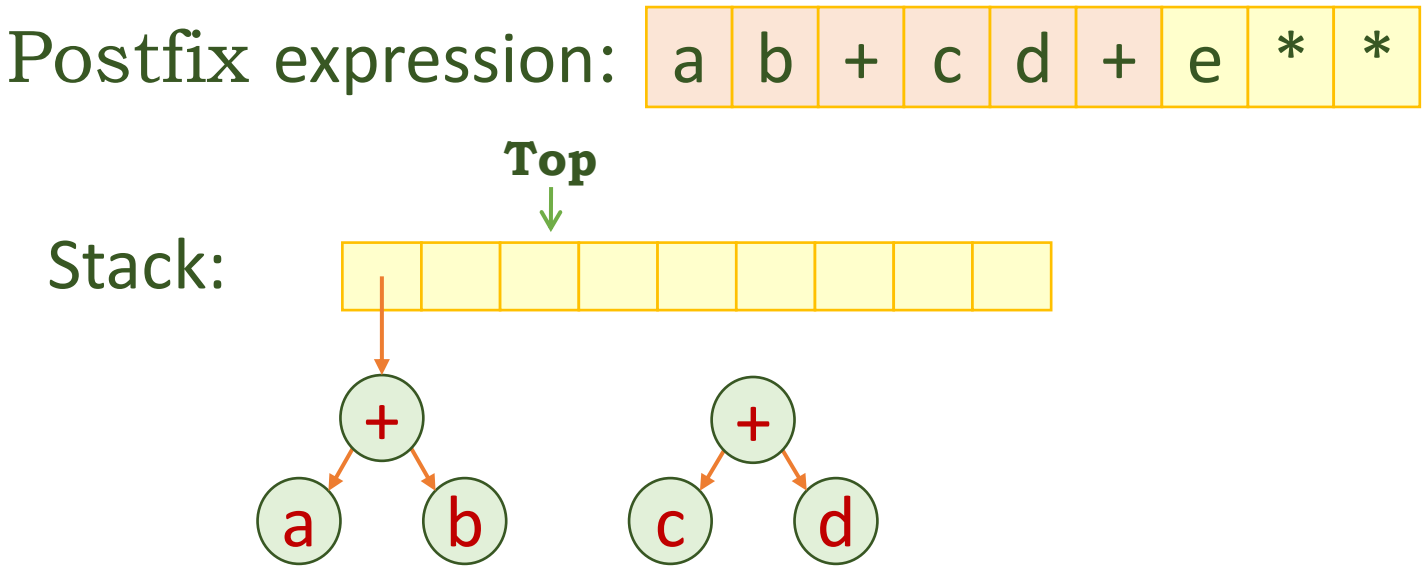
Postfix expression:

a	b	+	c	d	+	e	*	*

Stack:



Postfix expression and Stack

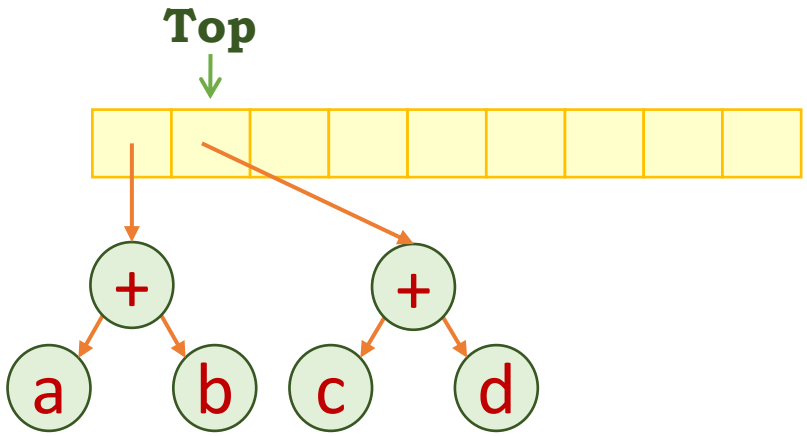


Postfix expression and Stack

Postfix expression:

a	b	+	c	d	+	e	*	*

Stack:

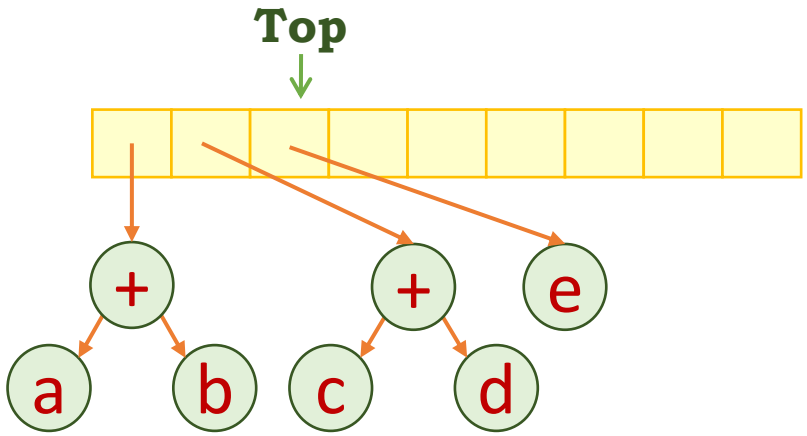


Postfix expression and Stack

Postfix expression:

a	b	+	c	d	+	e	*	*

Stack:

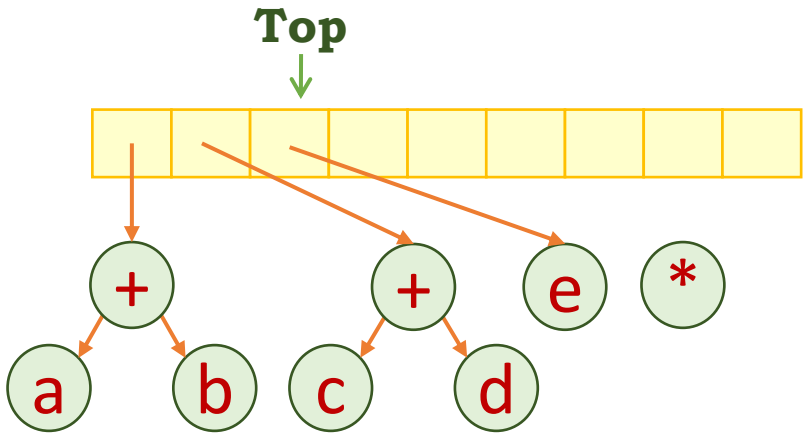


Postfix expression and Stack

Postfix expression:

a	b	+	c	d	+	e	*	*

Stack:

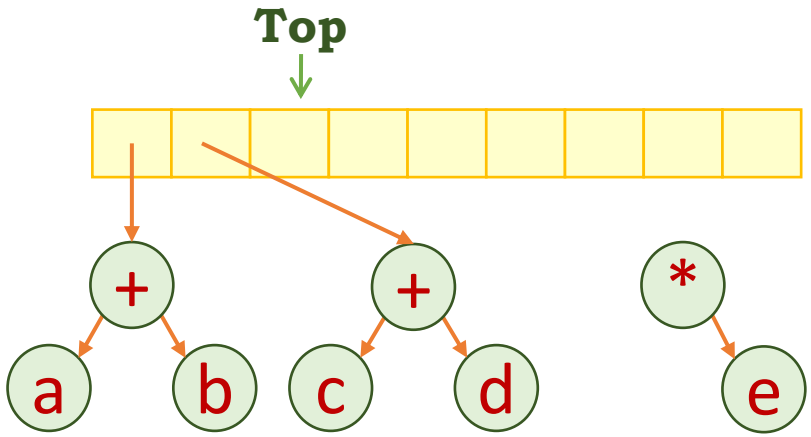


Postfix expression and Stack

Postfix expression:

a	b	+	c	d	+	e	*	*

Stack:

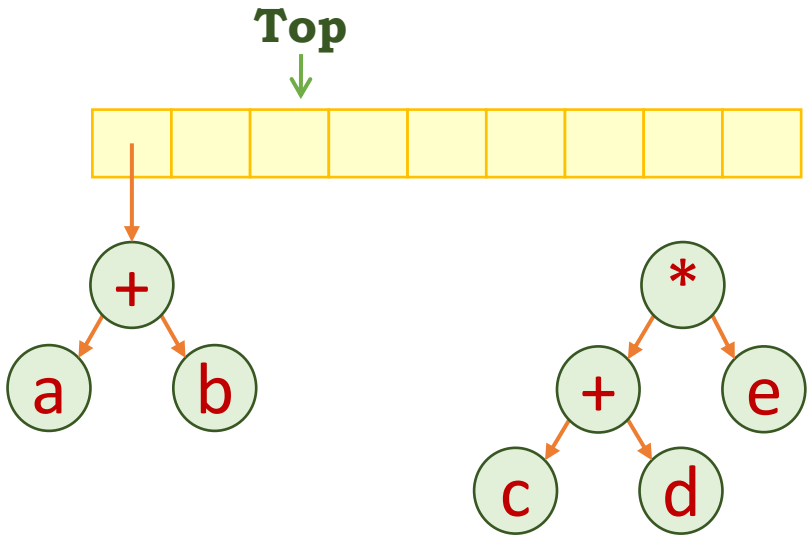


Postfix expression and Stack

Postfix expression:

a	b	+	c	d	+	e	*	*

Stack:

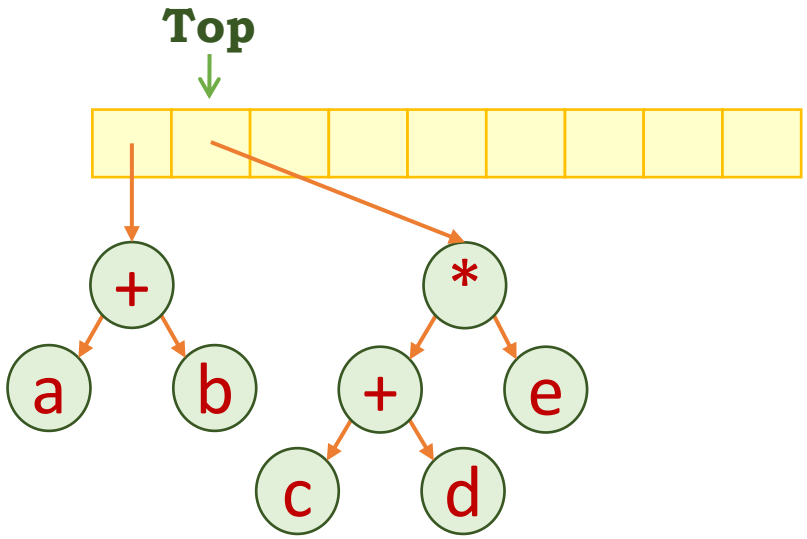


Postfix expression and Stack

Postfix expression:

a	b	+	c	d	+	e	*	*

Stack:

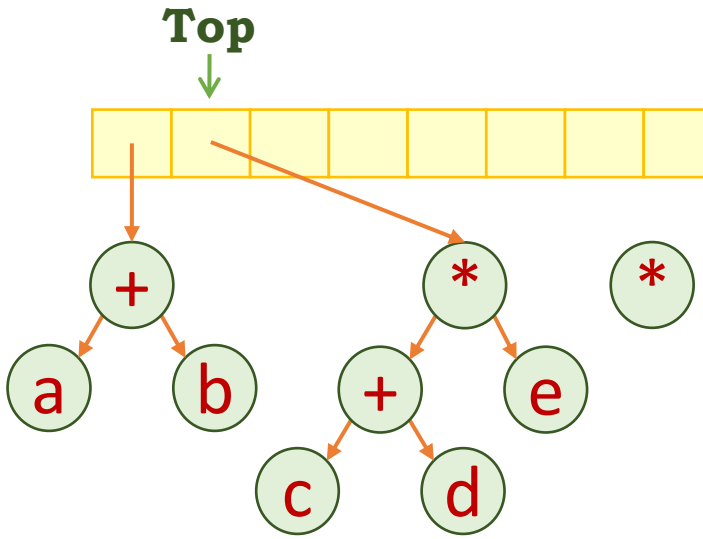


Postfix expression and Stack

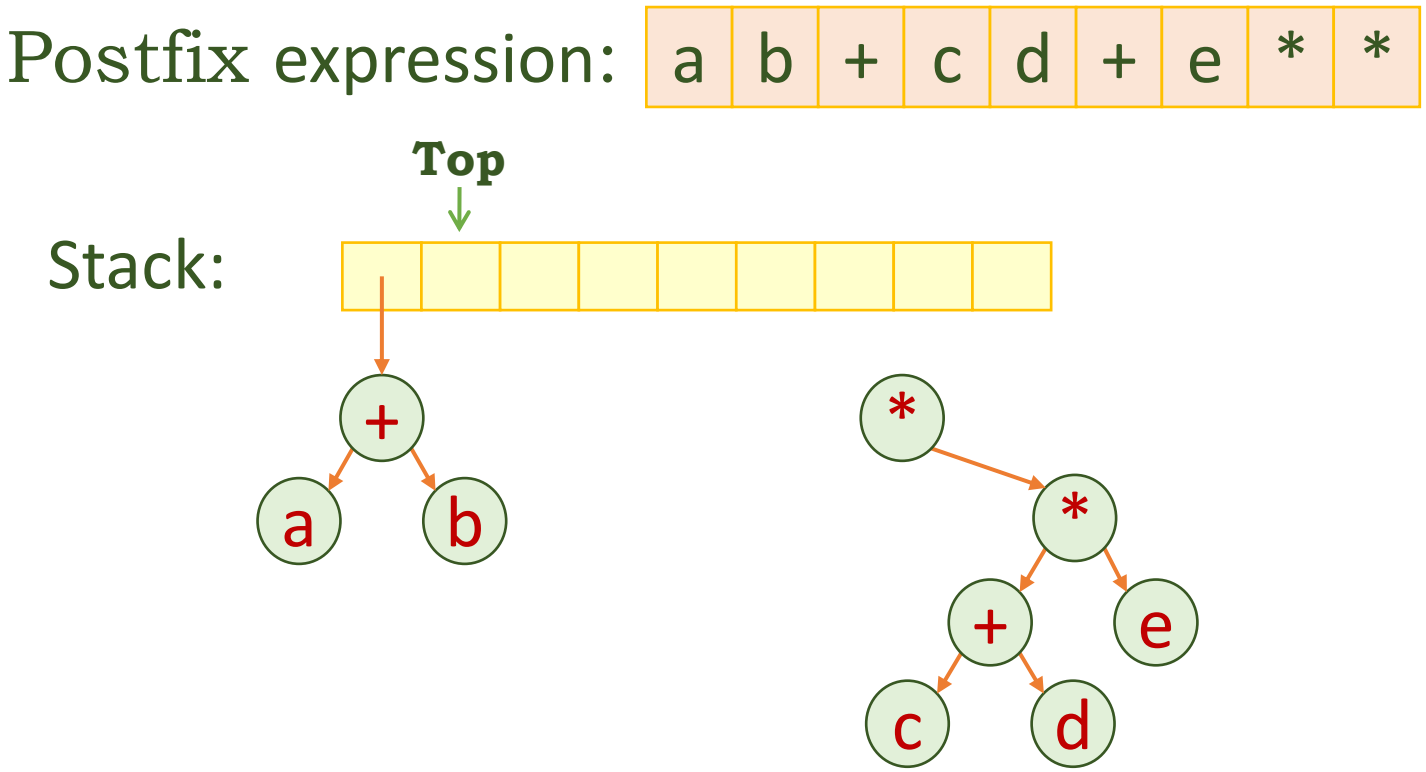
Postfix expression:

a	b	+	c	d	+	e	*	*

Stack:



Postfix expression and Stack



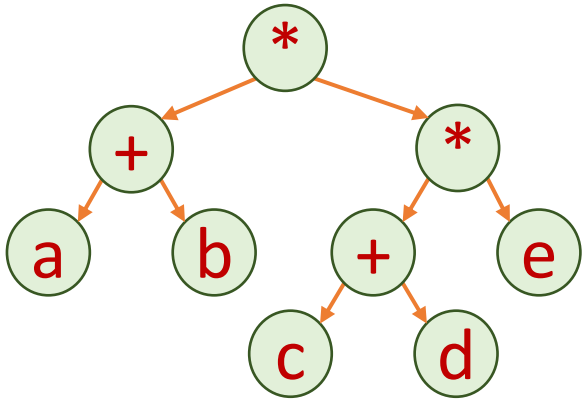
Postfix expression and Stack

Postfix expression:

a	b	+	c	d	+	e	*	*

Top

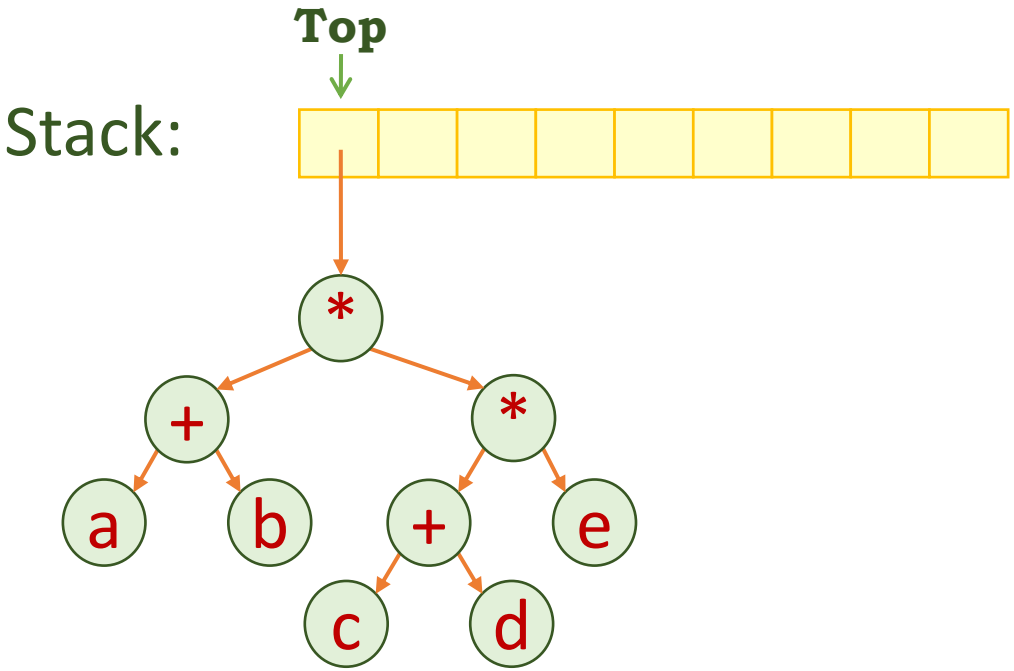
Stack:



Postfix expression and Stack

Postfix expression:

a	b	+	c	d	+	e	*	*




Binary Search
Tree

Binary Search Tree

A Binary search tree is:

- A binary tree
- The value in each node is greater than or equal to all the values in its left child or any of that child's descendants
- The value in each node is less than all the values in its right child or any of that child's descendants

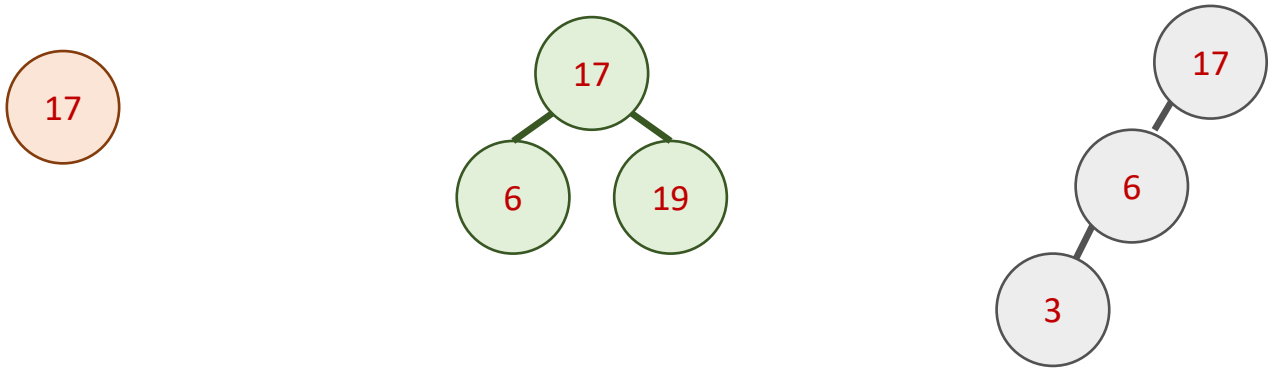
Binary Search Tree

Binary Search Trees

Binary Search Tree

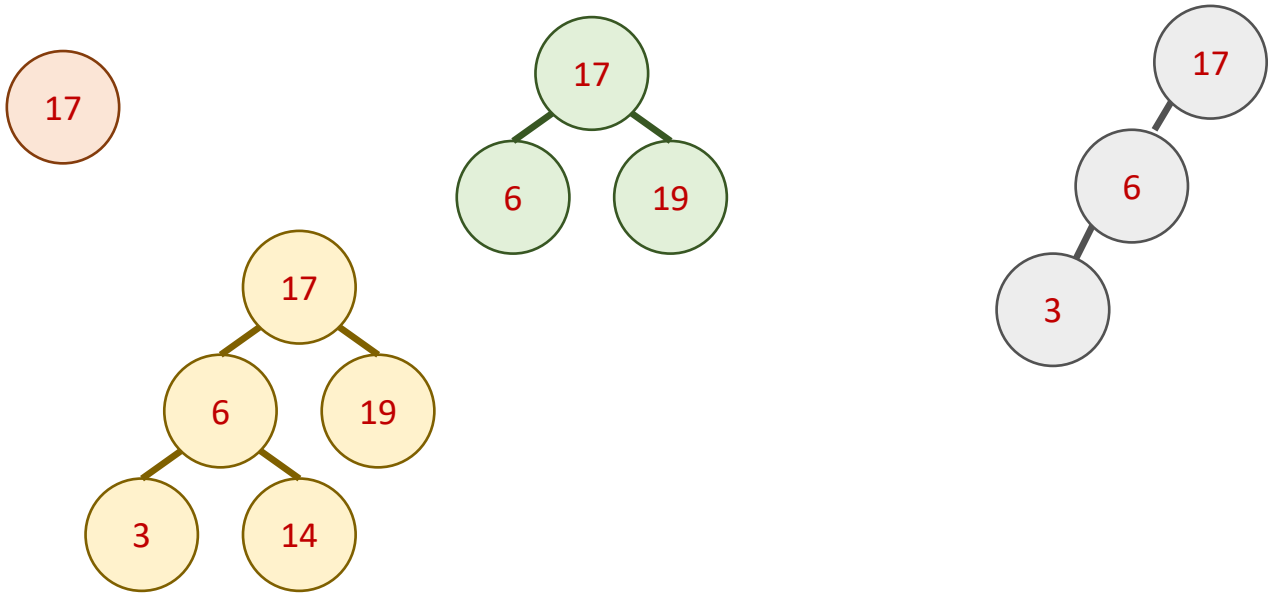
Binary Search Trees

Binary Search Tree



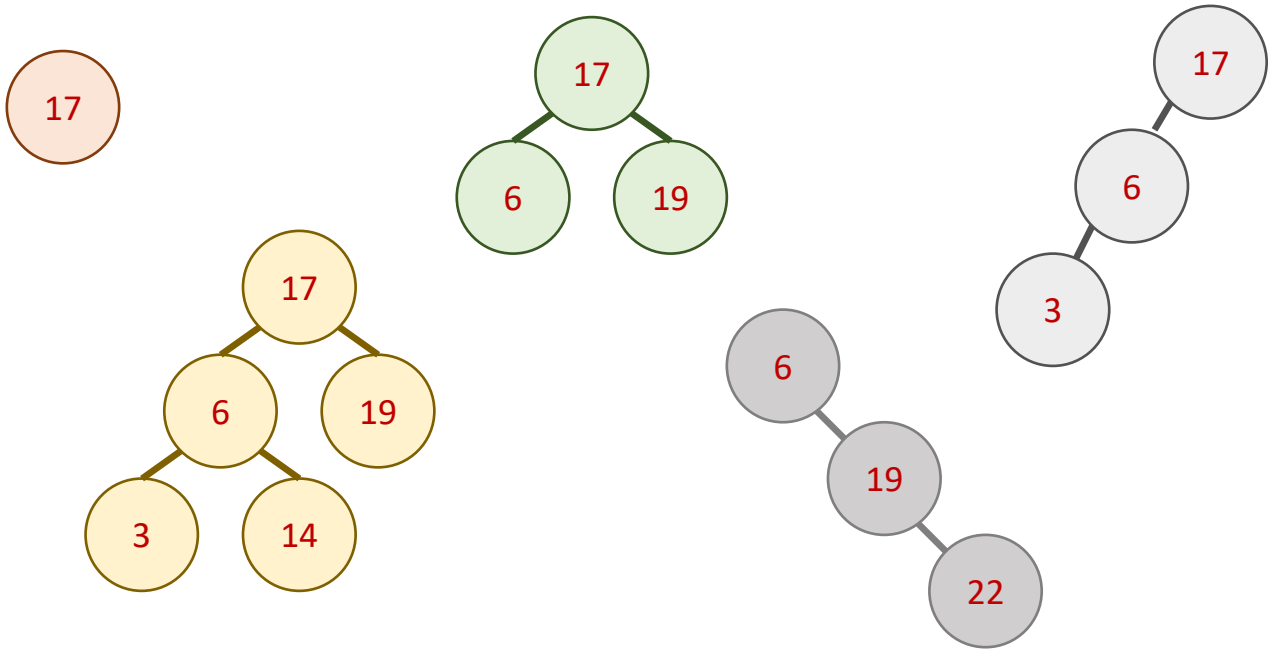
Binary Search Trees

Binary Search Tree



Binary Search Trees

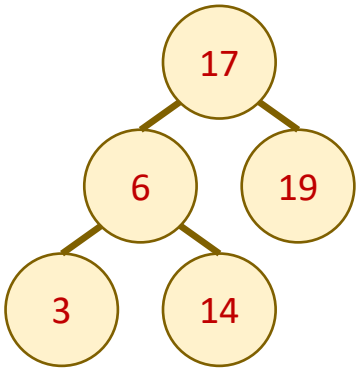
Binary Search Tree



Binary Search Trees

Binary Search Tree

- The inorder traversal of a binary search tree produces an ordered list.



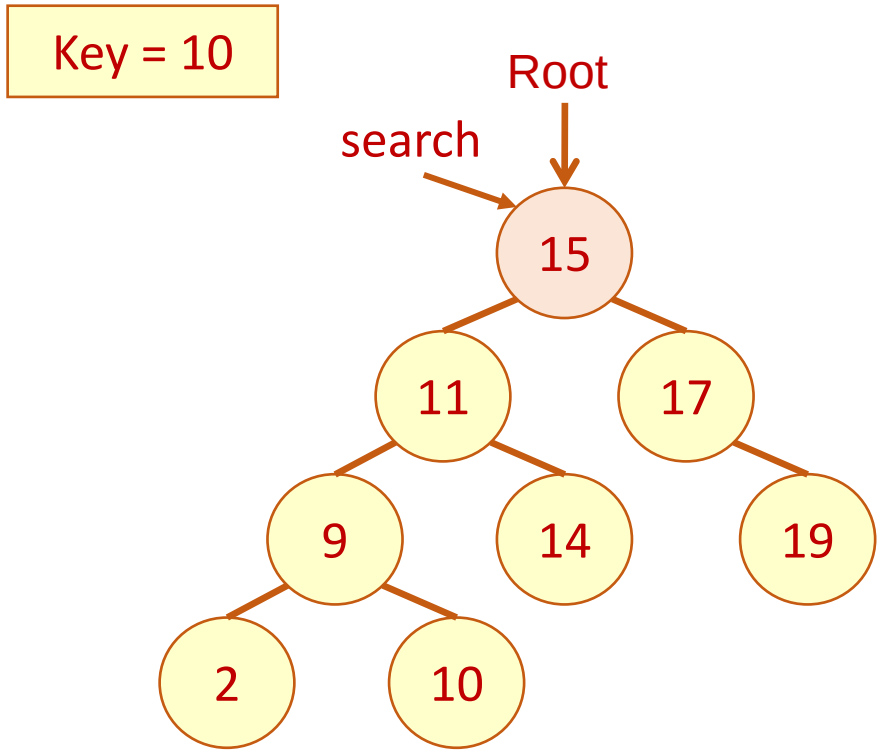
Inorder: 3, 6, 14, 17, 19

Binary Search Tree

Search

Problem:

- Given a “key” and a pointer “search” to the root
- Return the pointer to the node whose value is equal to key; return nil if there is no match

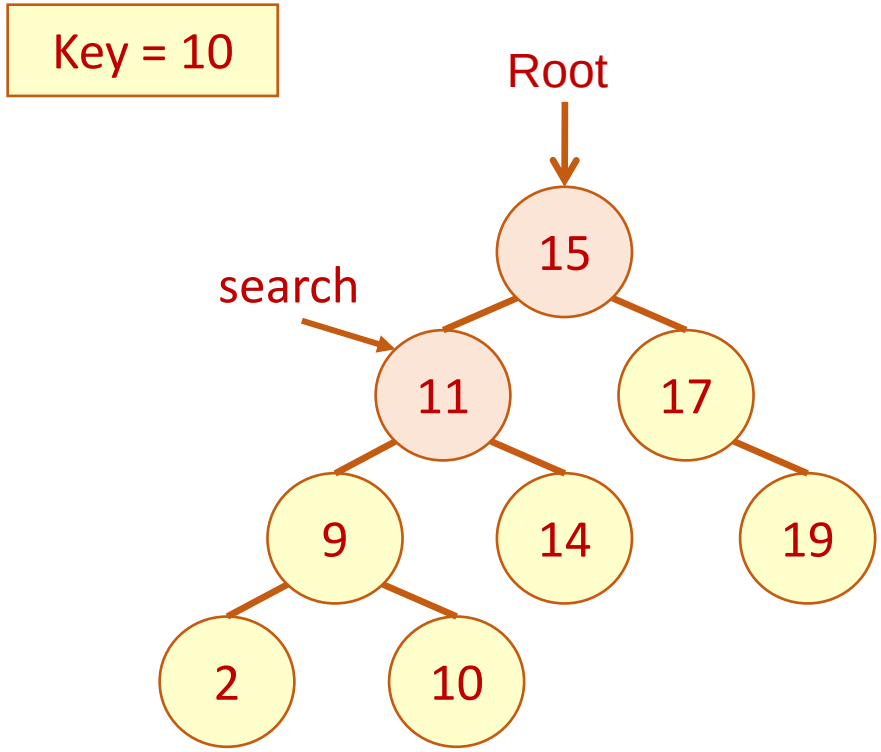


Binary Search Tree

Search

Problem:

- Given a “key” and a pointer “search” to the root
- Return the pointer to the node whose value is equal to key; return nil if there is no match

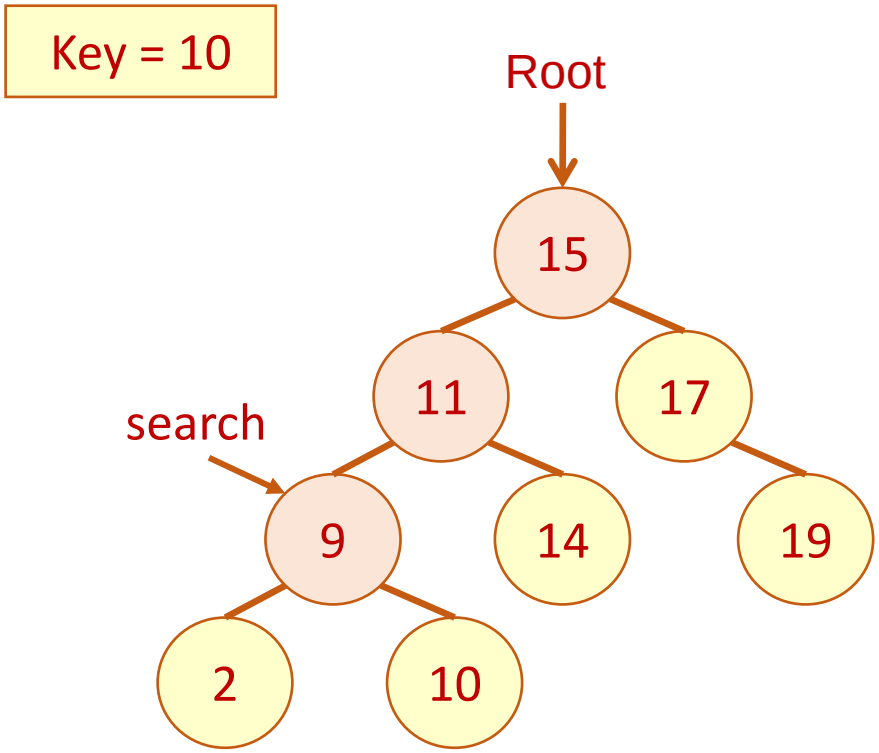


Binary Search Tree

Search

Problem:

- Given a “key” and a pointer “search” to the root
- Return the pointer to the node whose value is equal to key; return nil if there is no match

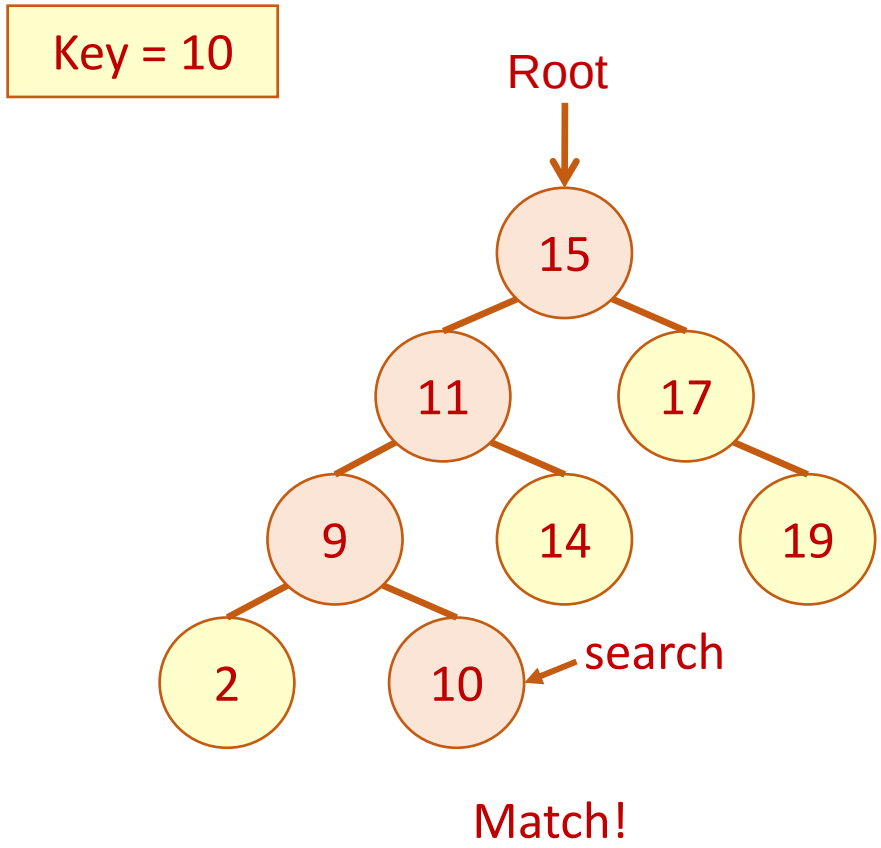


Binary Search Tree

Search

Problem:

- Given a “key” and a pointer “search” to the root
- Return the pointer to the node whose value is equal to key; return nil if there is no match



Binary Search Tree

Idea:

- BST property
- Compare key with the $search \rightarrow value$
  - if $key == (search \rightarrow value) \implies match!$
  - if $key < (search \rightarrow value)$
    - Search for the node on the $left$ sub-tree
    - else search for the node on the $right$ sub-tree

Binary Search Tree

Algorithm searchBST (val root <pointer>, val argument <key>)
Search a binary search tree for a given value.
Pre: root is the root to a binary tree or subtree argument is the key value requested
Return: the node address if the value is found null if the node is not in the tree

1. **If (root is null)**
   Return null
2. **End if**
3. **If (argument < root.key)**
   Return searchBST (root.left, argument)
4. **Elseif (argument > root.key)**
   Return searchBST (root.right, argument)
5. **Else**
   Return root
6. **End if**

End searchBST

Search for key in the left sub-tree.

Search for key in the right sub-tree.

Binary Search Tree

Insert

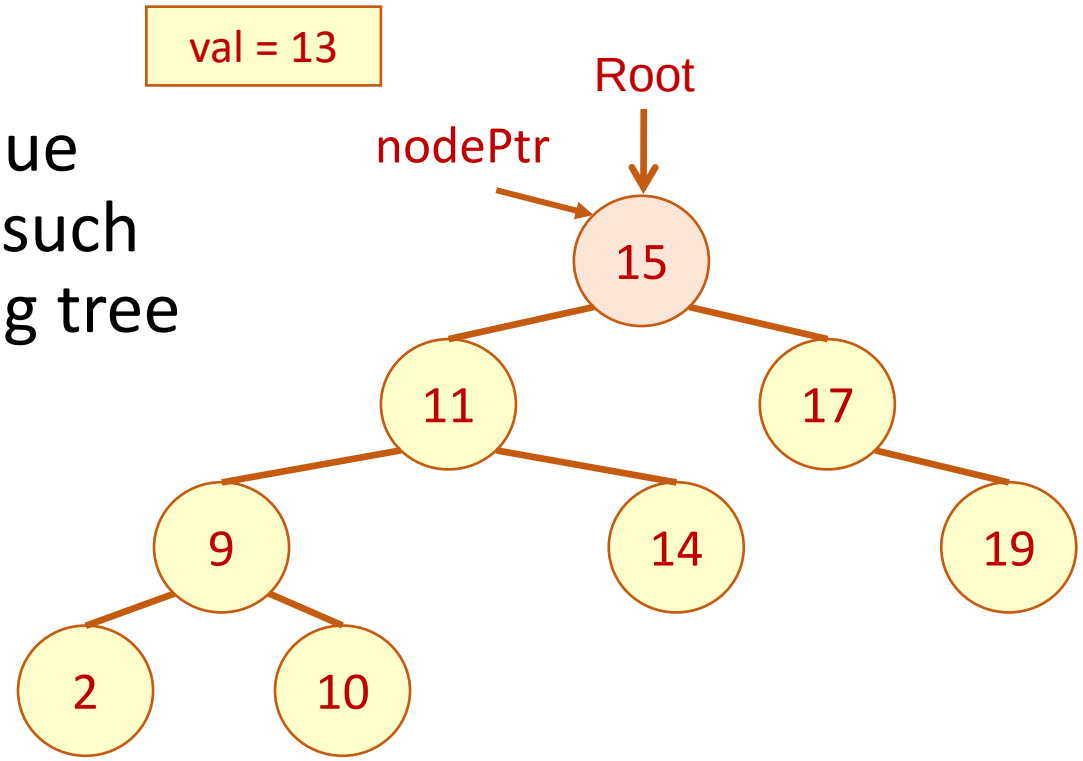
- Inserting data into a binary search tree is simple.
  - Follow the branches to an empty subtree and then insert the new node.

All BST inserts take place at a leaf or a leaf-like node, that is, a node that has only one null branch.

Binary Search Tree

Problem

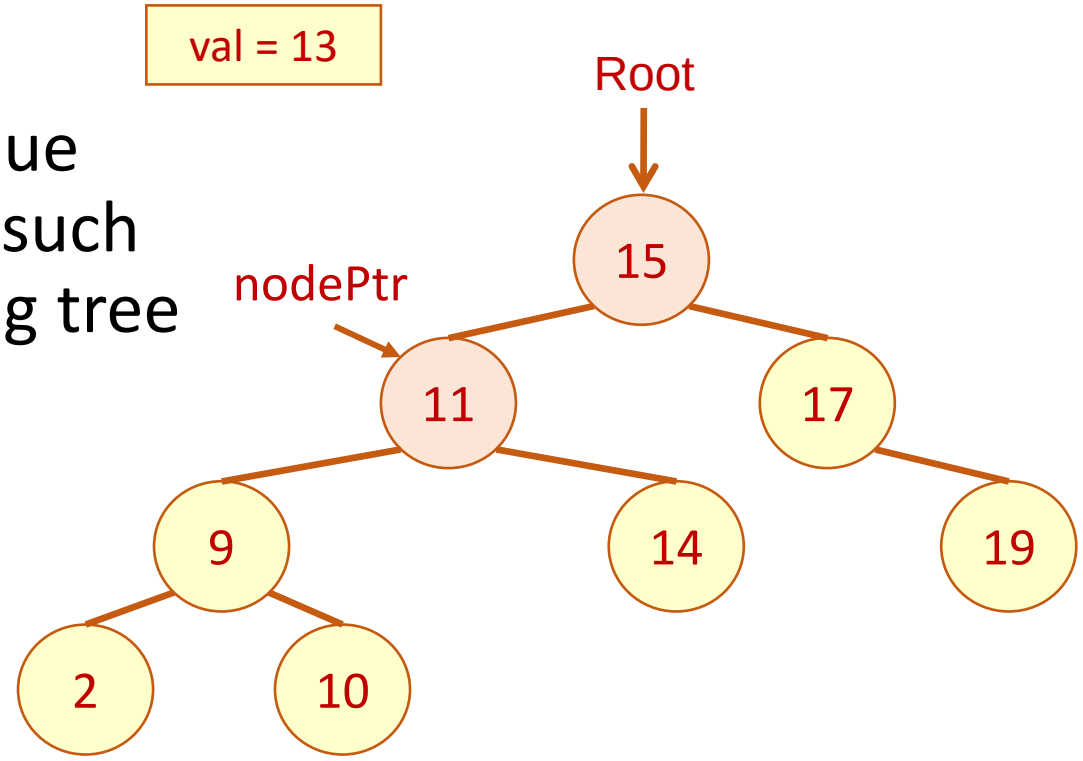
- Inserting the value *"val"* into a BST such that the resulting tree is also a BST



Binary Search Tree

Problem

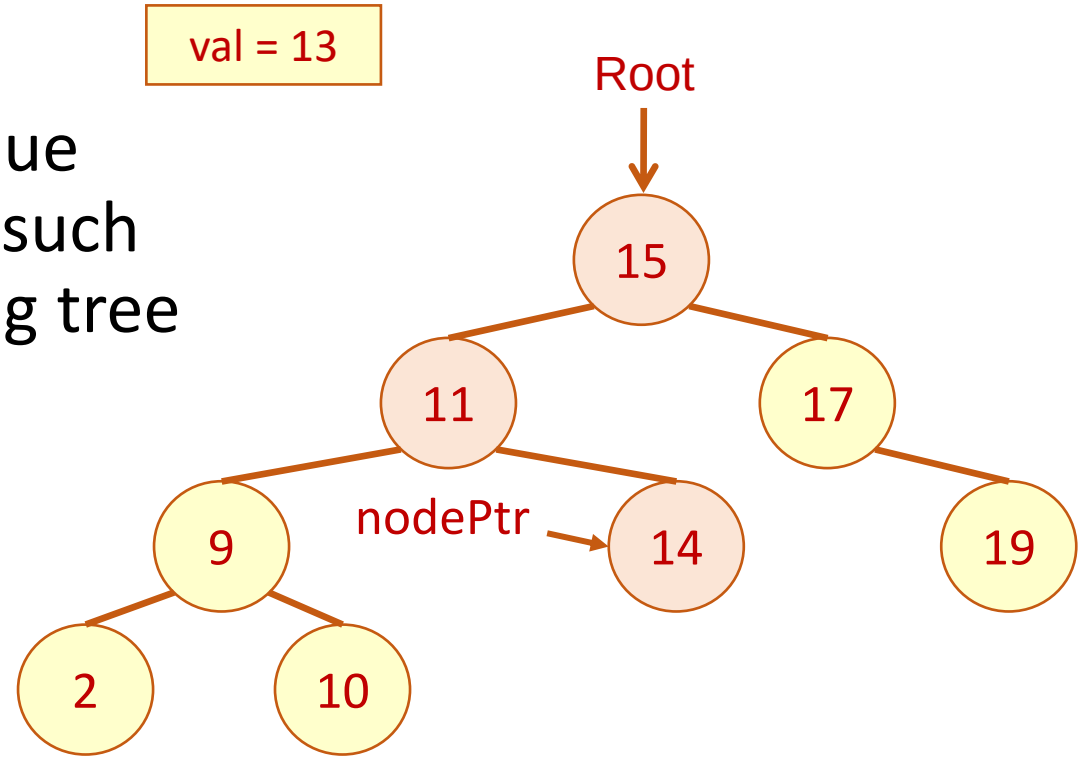
- Inserting the value *"val"* into a BST such that the resulting tree is also a BST



Binary Search Tree

Problem

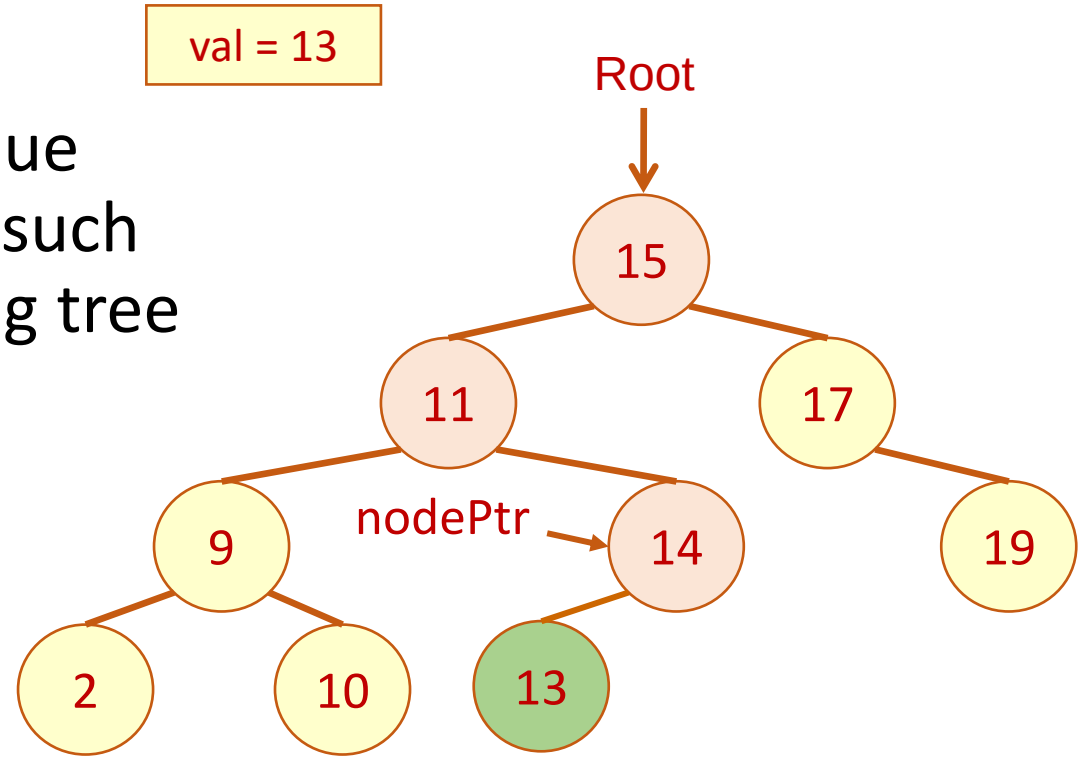
- Inserting the value *"val"* into a BST such that the resulting tree is also a BST



Binary Search Tree

Problem

- Inserting the value *"val"* into a BST such that the resulting tree is also a BST



Binary Search Tree

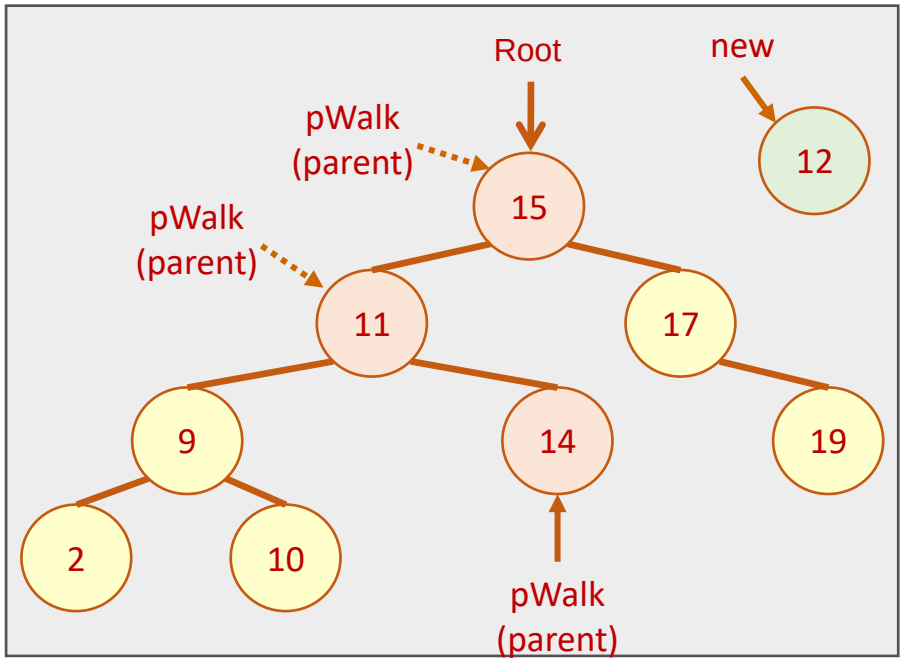
Idea:

- Compare *val with tree.value*
  - If *val < tree.value, val* should be inserted into the left sub-tree
  - Otherwise, insert *val* into the right sub-tree

Binary Search Tree

Algorithm insertBST (ref root <pointer>, val new <pointer>)
Insert node containing new node into BST using iteration.
Pre: root is address of first node in a BST
new is address of node containing data to be inserted
Post: new node inserted into the tree
If (root is null)
Root = new
else

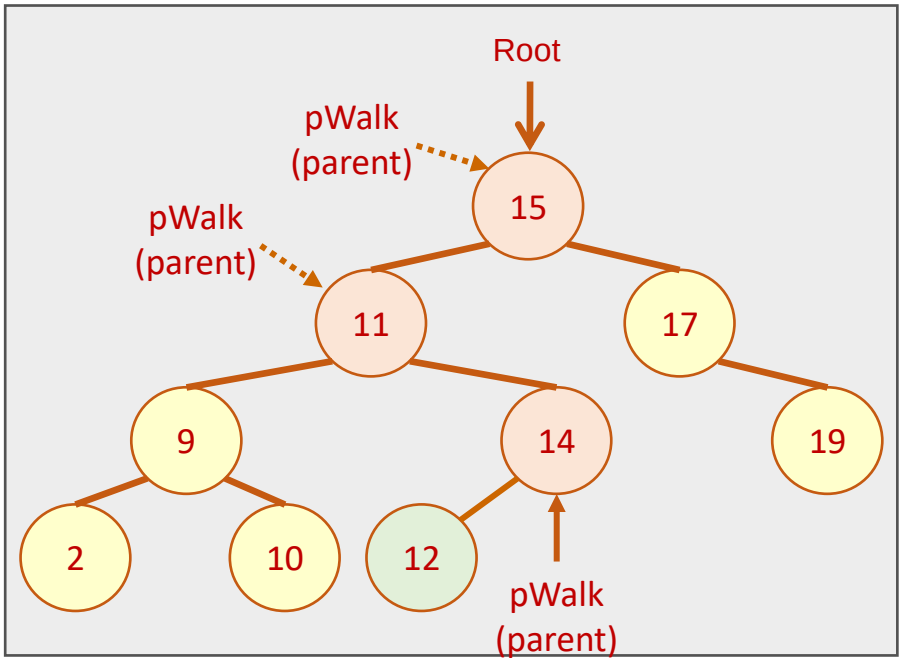
Binary Search Tree




```
pWalk = root
Loop (pWalk not null)
   parent = pWalk
   If (new.key < pWalk.key)
      pWalk = pWalk.left     // follow to the left
   Else
      pWalk = pWalk.right    // follow to the right
   End if
End loop
```

Binary Search Tree

```
    // Location for new node found (parent)
  If (new.key < parent.key)
    parent.left = new
  Else
    parent.right = new
  End if
End if
Return
End insertBST
```



Binary Search Tree

Algorithm addBST (ref root <pointer>, val new <pointer>)

Insert node containing new data into BST using recursion.

Pre: root is address of current node in a BST
new is address of node containing data to be inserted

Post: new node inserted into the tree

If (root is null)
  Root = new
else

Binary Search Tree

```
   // Locate null subtree for insertion
   If (new.key < root.key)
      addBST (root.left, new)
   Else
      addBST (root.right, new)
   End if
End if
Return
End addBST
```

Binary Search Tree

Delete

To delete a node from a binary search tree, we first locate the node. There are four possible cases when we delete a node:

1. The node to be deleted has no children. In this case, we can just delete the node by setting the node's parent to null, recycle its memory and return.

Binary Search Tree

2. The node to be deleted has only a right subtree. If there is only a right subtree, then we can simply attach the right subtree to the delete node's parent and recycle its memory.
3. The node to be deleted has only a left subtree. If there is only a left subtree, then we attach the left subtree to the delete node's parent and recycle its memory.

Binary Search Tree

4. The node to be deleted has two subtrees. In order to maintain the existing structure as much as possible, we can do the one of the following:
   1. Find the largest node in the deleted node's left subtree and move its data to replace the deleted node's data or
   2. Find the smallest node on the deleted node's right subtree and move its data to replace the deleted node's data.

Binary Search Tree

Algorithm deleteBST (ref root <pointer>, val delKey <key>)

This algorithm deletes a node from a BST.

Pre: root is pointer to tree containing data to be deleted.
delKey is key of node to be deleted

Post: node deleted and memory recycled.
if delKey not found, root unchanged

Return: true if node deleted, false if not found

Binary Search Tree

```
If (root is null)
    Return false
End if
If (delKey < root.data.key)
    Return deleteBST (root.left, delKey)
Elseif (delKey > root.data.key)
    Return deleteBST (root.right, delkey)
else
```

Binary Search Tree

Delete node found – test for leaf node
If (root.left is null)
    delPtr = root
    root = root.right
    Recycle (delPtr)
    Return true
Elseif (root.right is null)
    delPtr = root
    root = root.left
    Recycle (delPtr)
    Return true
else

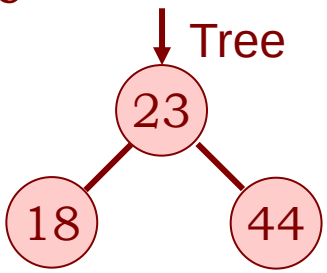
Binary Search Tree

```
        Node to be deleted is not a leaf. Find largest
        node on left subtree.
        delPtr = root.left
        Loop (delPtr.right not null)
            delPtr = delPtr.right
            Node found. Move data and delete leaf node.
        Root.data = delPtr.data
        Return deleteBST (root.left, delPtr.data.key)
    End if
End if
End deleteBST
```

Binary Search Tree

Delete node (44) that has no children.

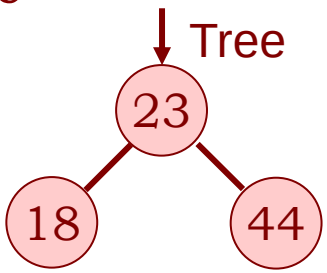
Before



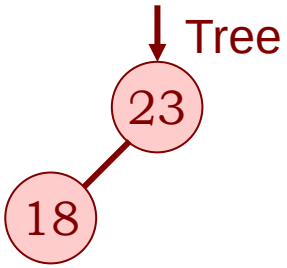
Binary Search Tree

Delete node (44) that has no children.

Before



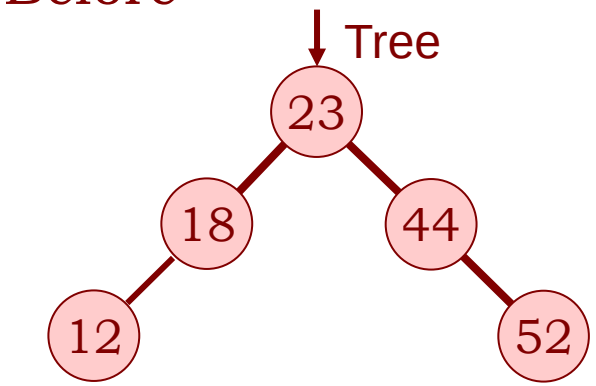
After



Binary Search Tree

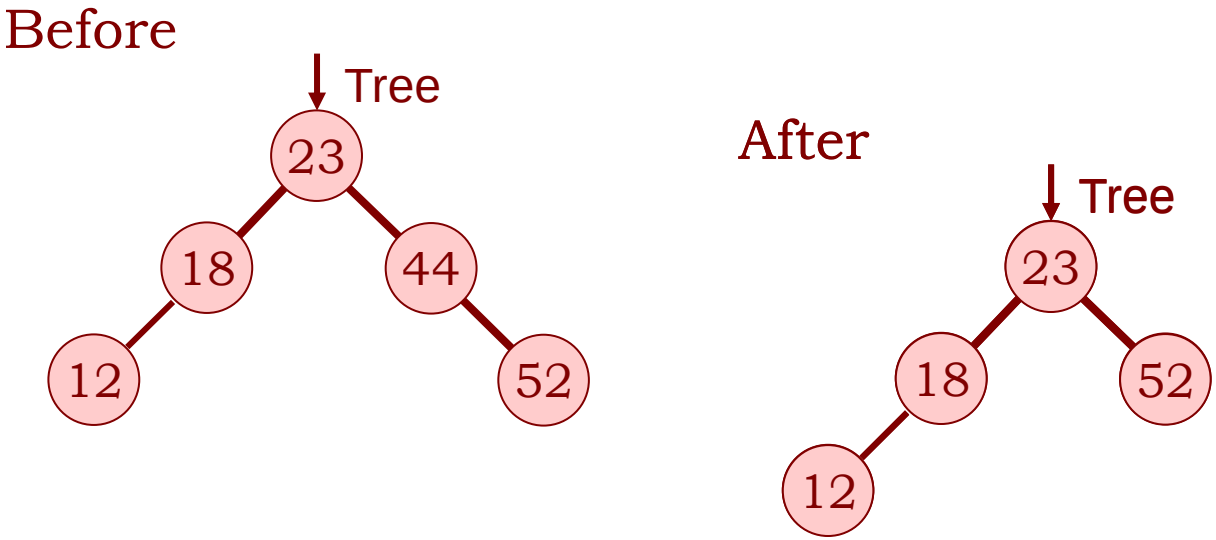
Delete node (44) with no left child

Before



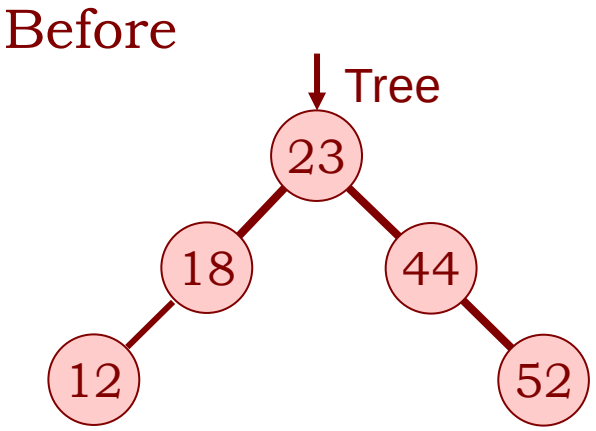
Binary Search Tree

Delete node (44) with no left child



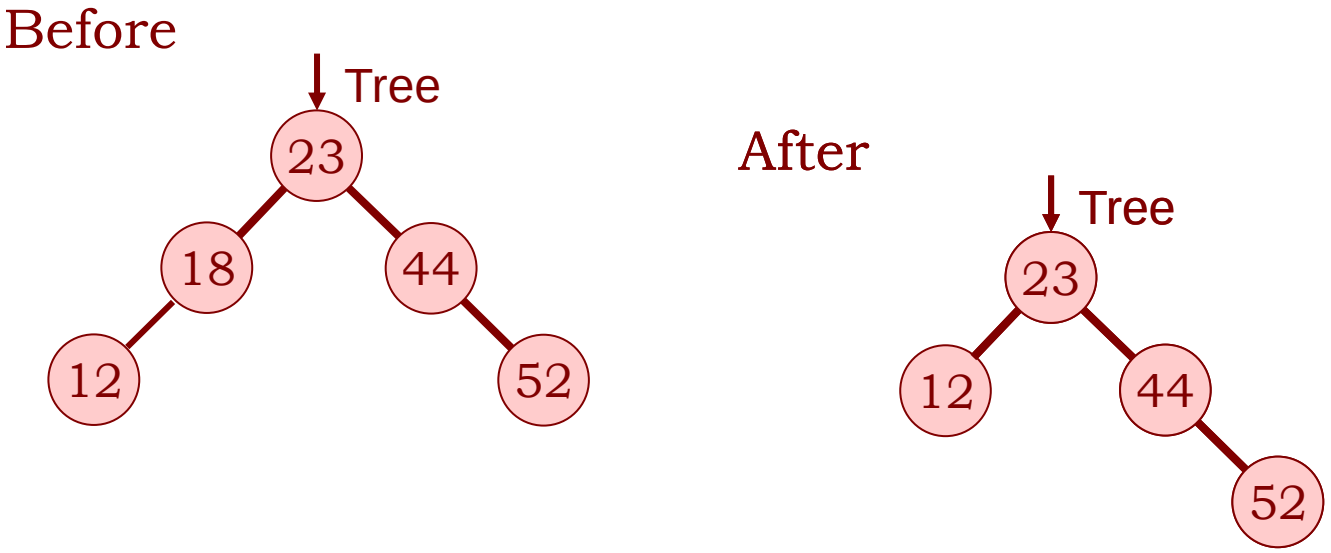
Binary Search Tree

Delete node (18) with no right child



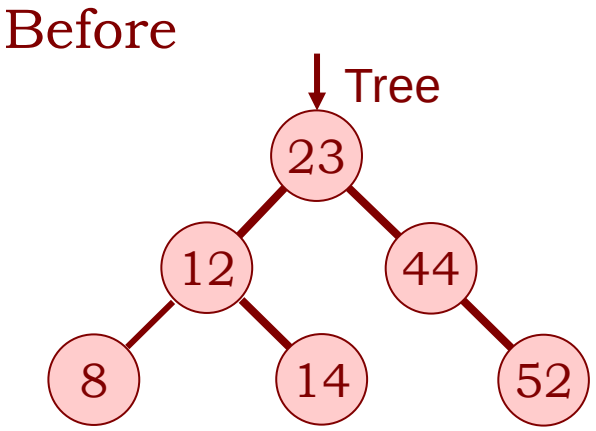
Binary Search Tree

Delete node (18) with no right child



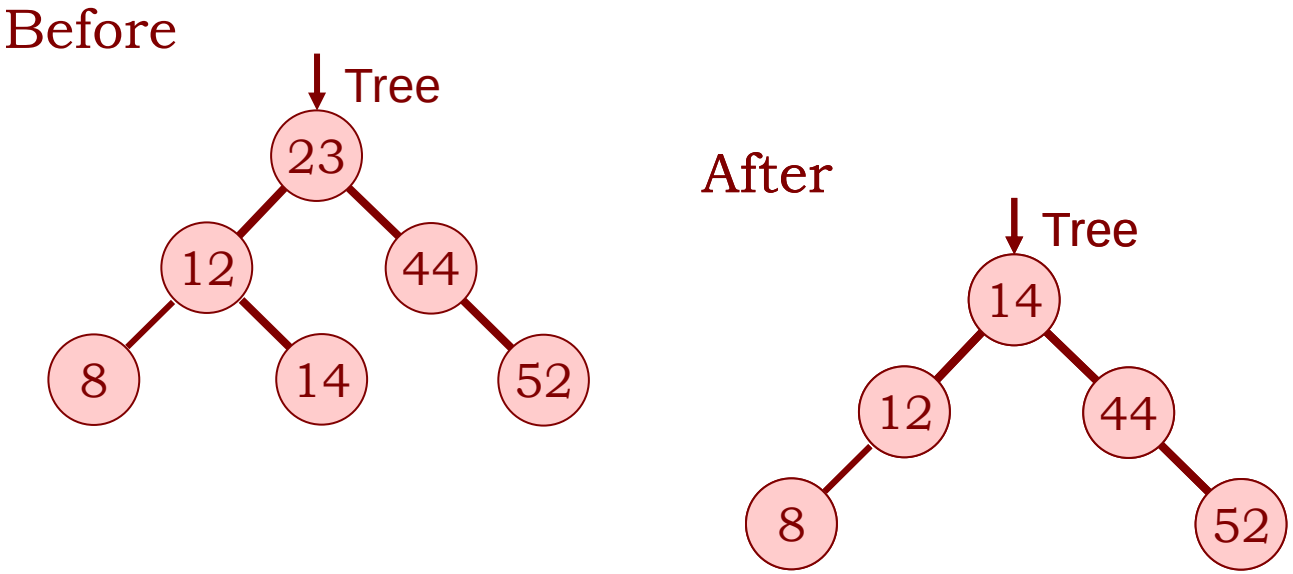
Binary Search Tree

Delete node 23 with two children



Binary Search Tree

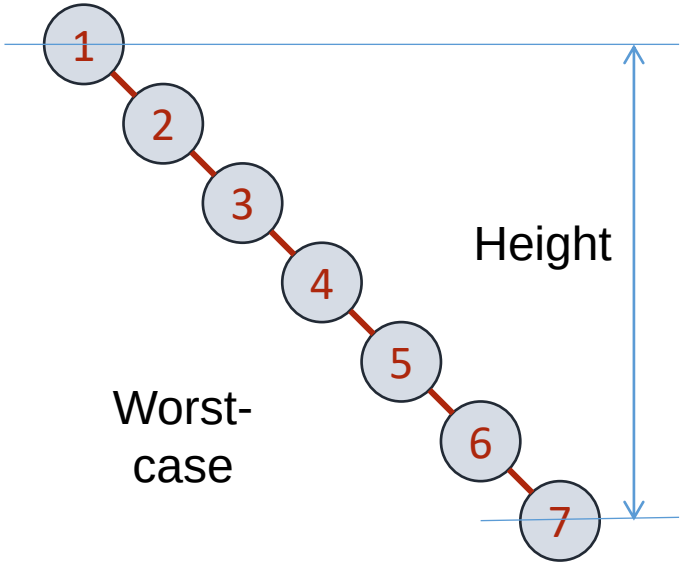
Delete node 23 with two children

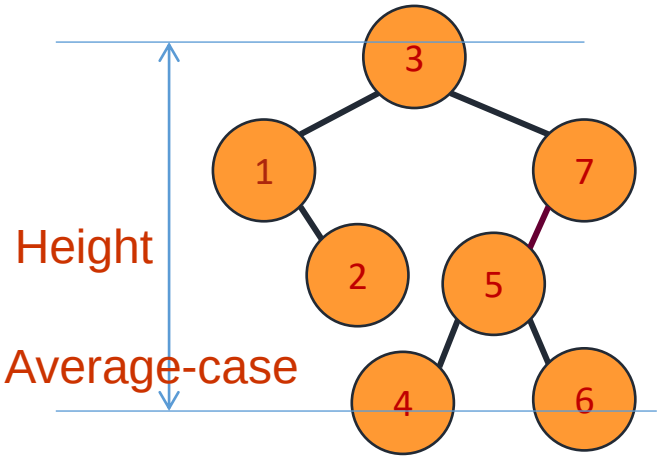


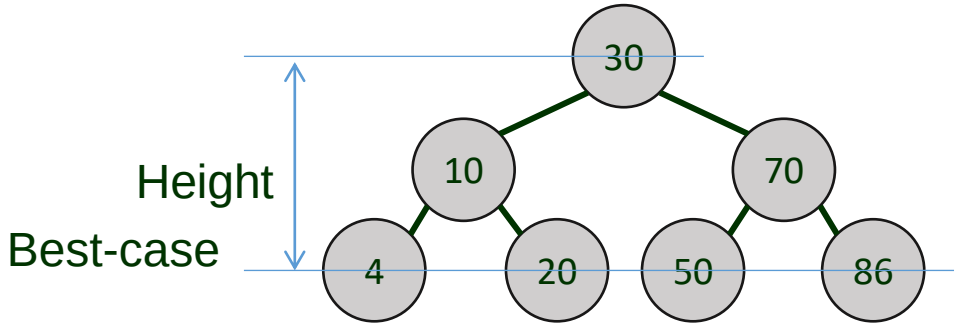
Running time complexity of Binary Search Tree

What is the running time complexity of Binary Search Tree?

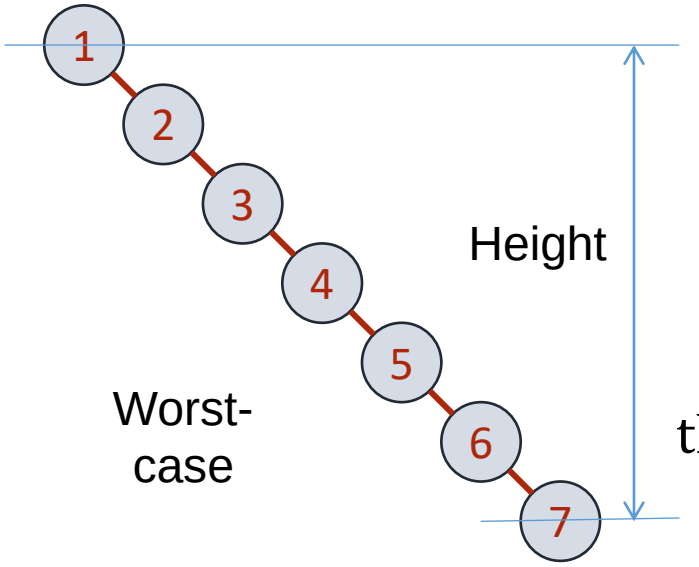
Worst-case, Best-case and Average-case BST







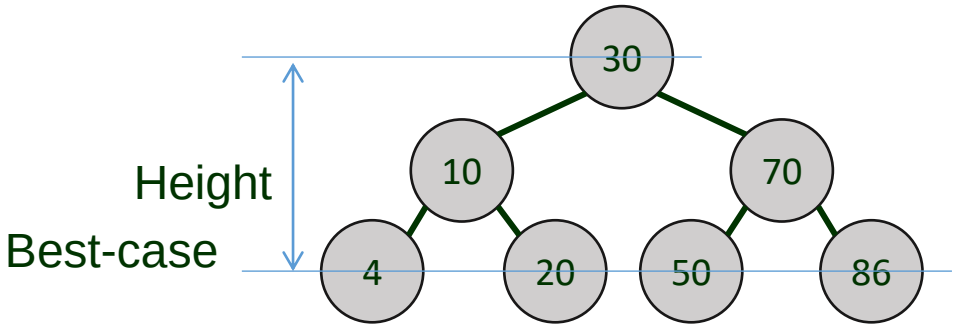
Worst-case, Best-case and Average-case BST



For a worst
− case scenario,

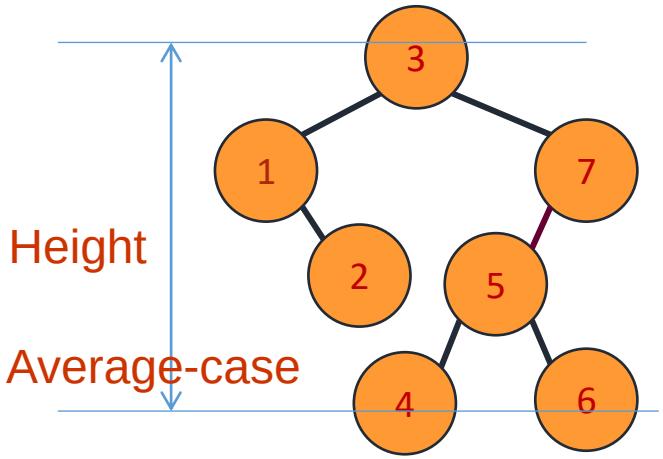
the height of the BST is n.

Worst-case, Best-case and Average-case BST



For a best − case scenario, the height of the BST is lg n.

Worst-case, Best-case and Average-case BST



For an average
– case scenario,
the height of the BST is lg n

Binary Search Tree Efficiency

- The disadvantage of a binary search **tree is that its height** can be as large as N-1
  - This means that the time needed to perform insertion and deletion and many other operations can be O(N) in the worst case
- We want a **tree with small height**
  - A binary **tree with N node has height at least $O(\log_2 N)$**
  - Thus, our goal is to keep the height of a binary search tree at **$O(\log_2 N)$**
- Such trees are called balanced binary search trees. Examples are **AVL tree, red-black tree.**

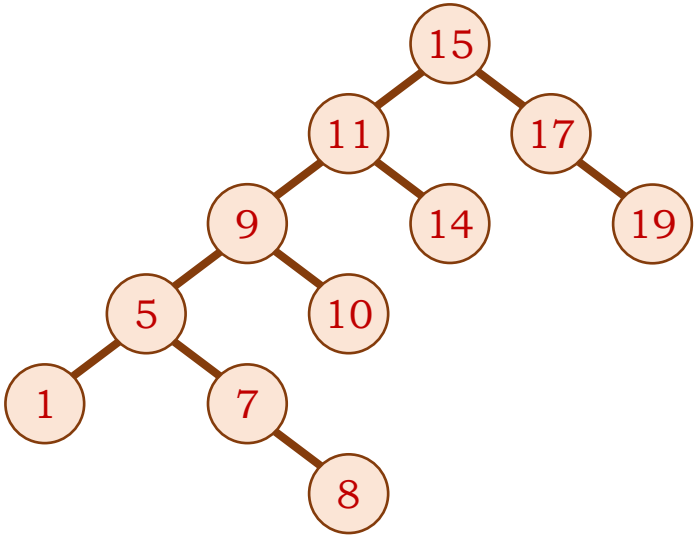

ADELSON-
VELSKI AND
LANDIS
(AVL) Tree

AVL Tree

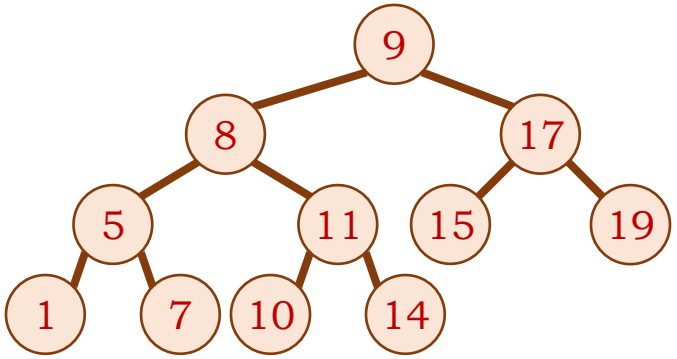
- An **AVL tree is a self-balancing binary search tree** in which
  - The heights of the left and right sub-trees of any node differ by at most one; therefore, it is also said to be height-balanced.

AVL Tree

Unbalanced BST



Balanced BST

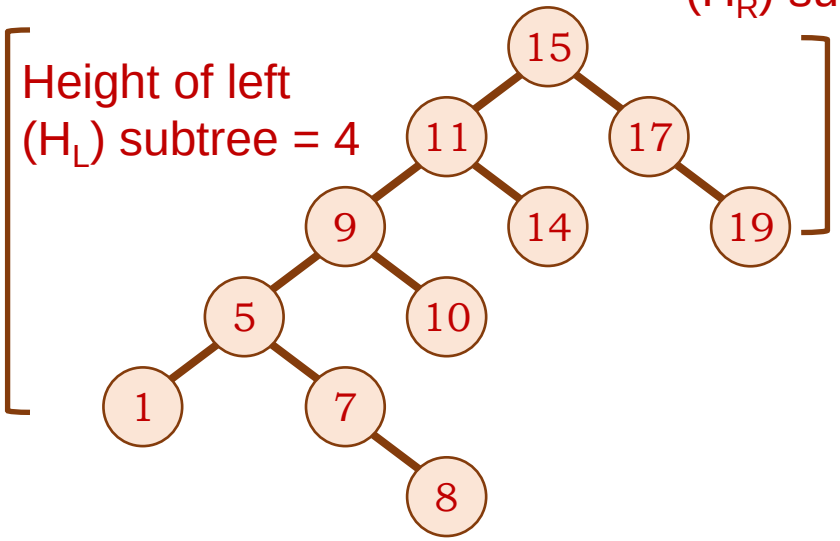


AVL Tree

Unbalanced BST

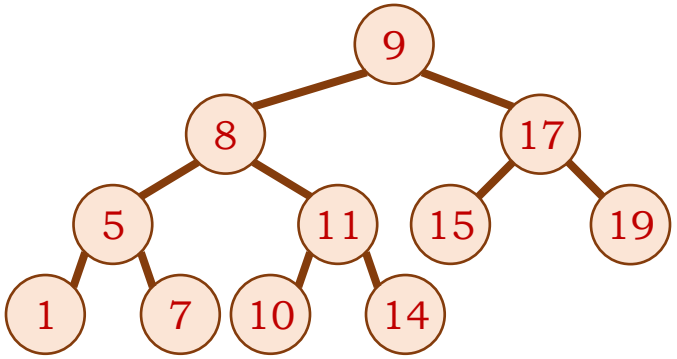
Height of right (H_R) subtree = 2

Height of left (H_L) subtree = 4



$|H_L - H_R| \geq 1, \Longrightarrow$ Unbalanced

Balanced BST

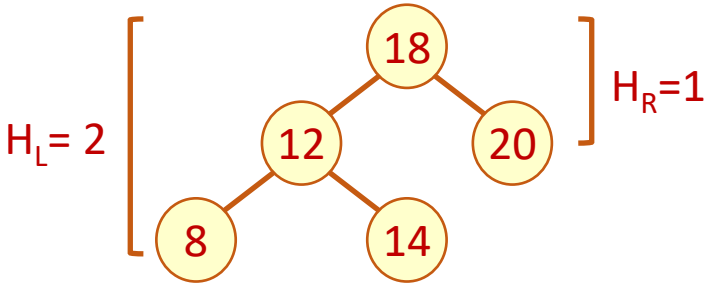


$|H_L - H_R| \leq 1, \Longrightarrow$ Balanced

AVL Tree

- Whenever we insert a node into or delete a node from a tree, the resulting tree may become unbalanced.

Before



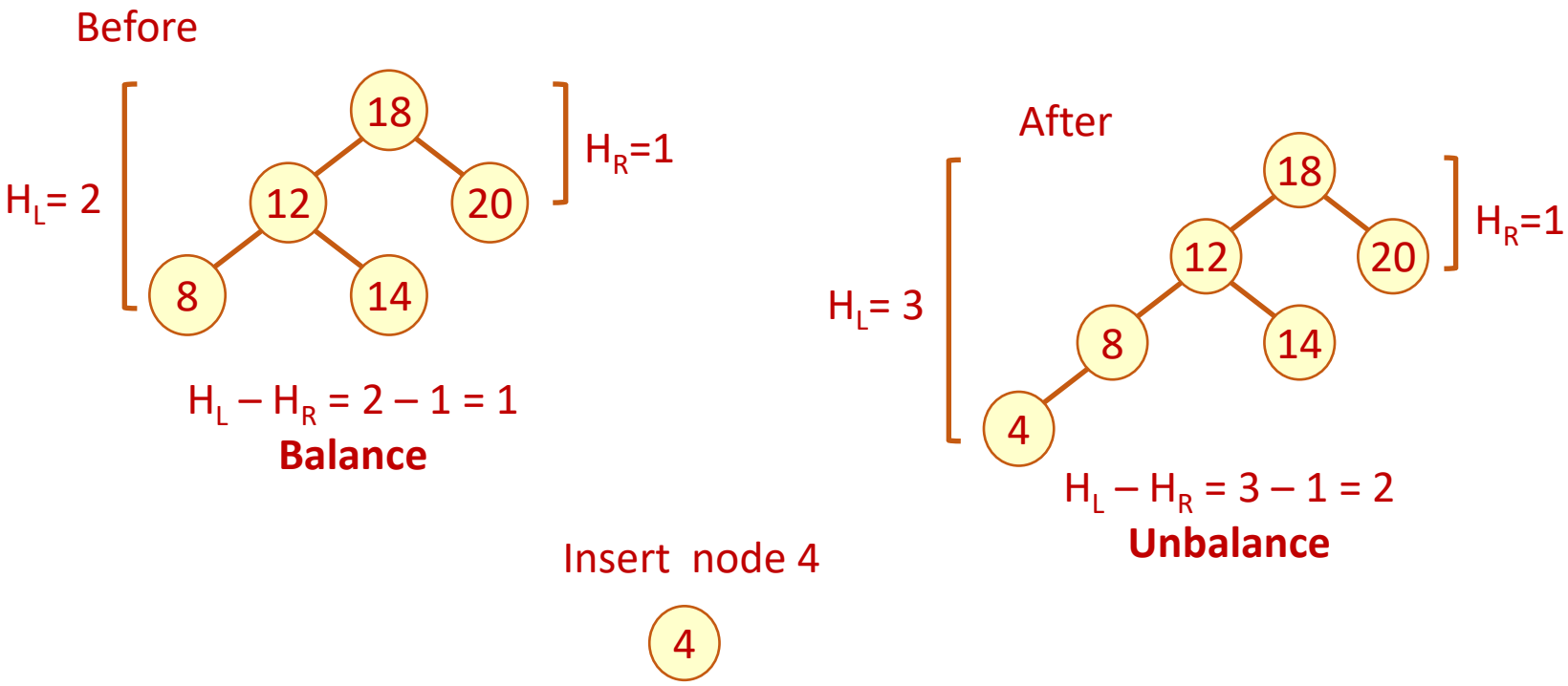
$H_L - H_R = 2 - 1 = 1$

Balance

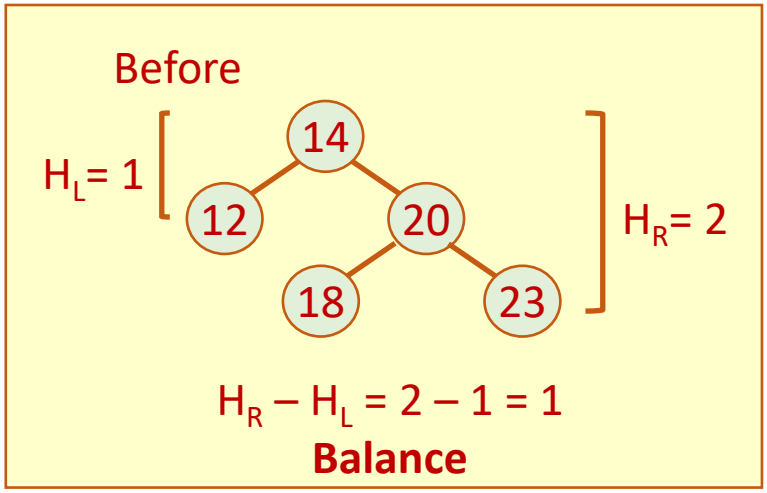
Insert node 4

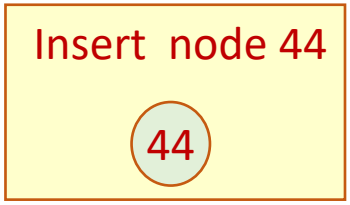
AVL Tree

- Whenever we insert a node into or delete a node from a tree, the resulting tree may become unbalanced.

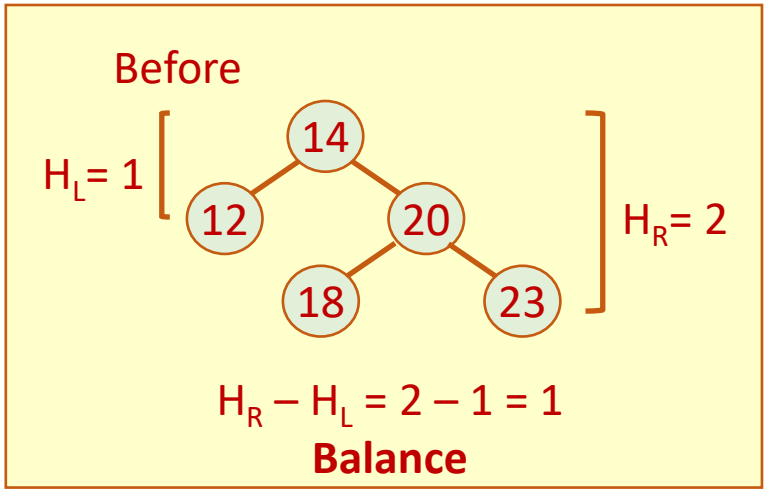


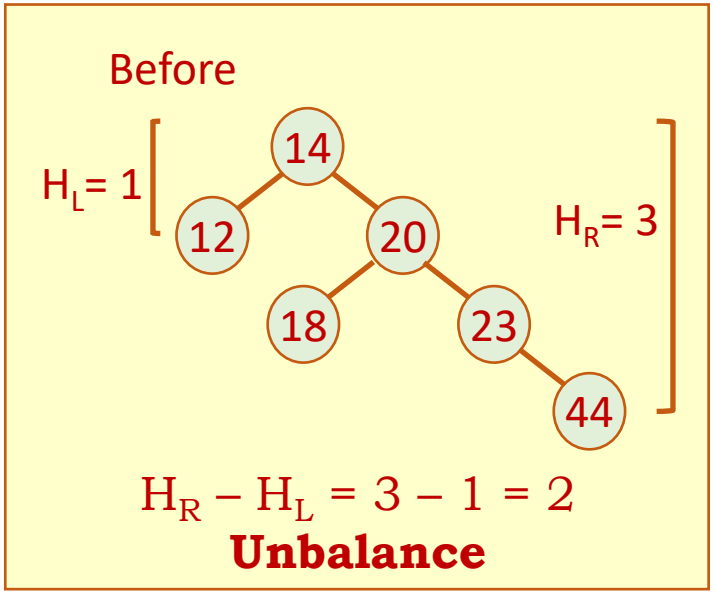
AVL Tree

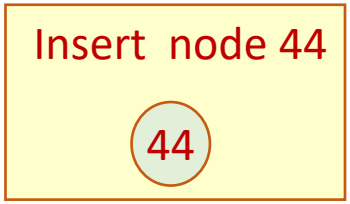

Before
H_L= 1
14
12
20
18
23
H_R= 2
$H_R - H_L = 2 - 1 = 1$
Balance


Insert node 44
44

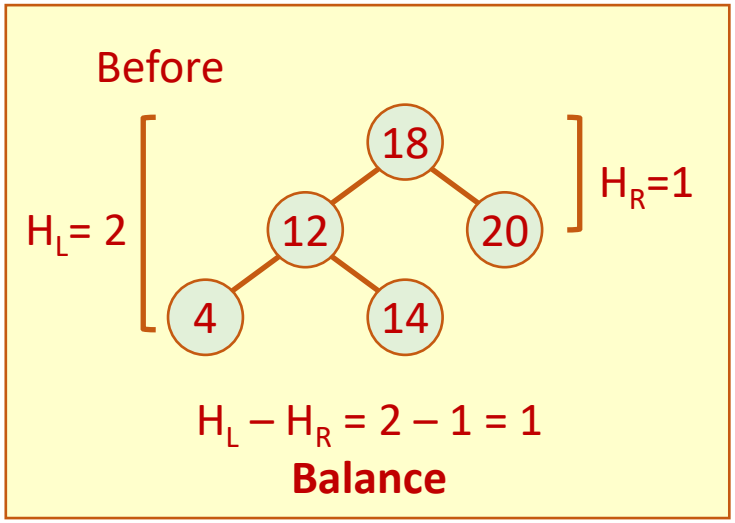
AVL Tree

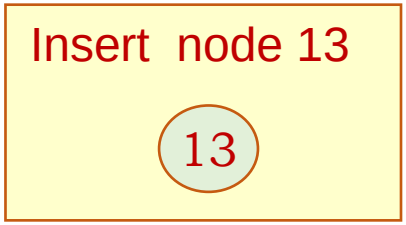

Before
H_L= 1
14
12
20
18
23
H_R= 2
$H_R - H_L = 2 - 1 = 1$
Balance


Before
H_L= 1
14
12
20
18
23
44
H_R= 3
$H_R - H_L = 3 - 1 = 2$
Unbalance

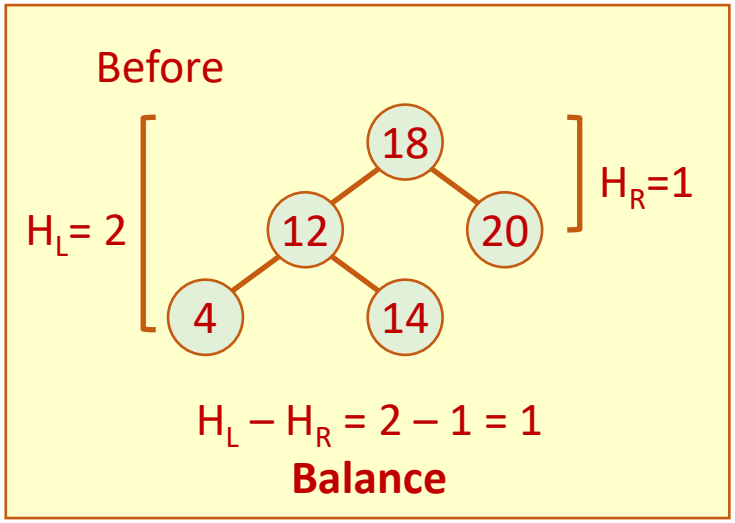

Insert node 44
44

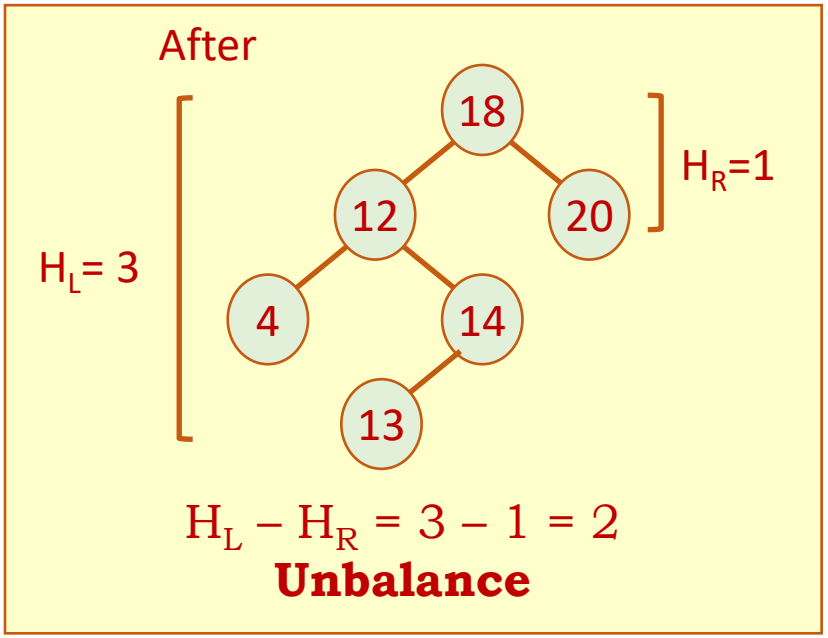
AVL Tree

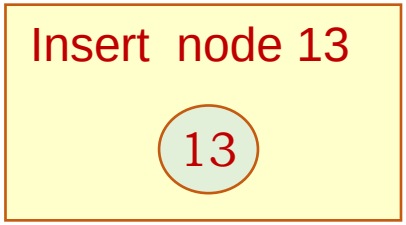




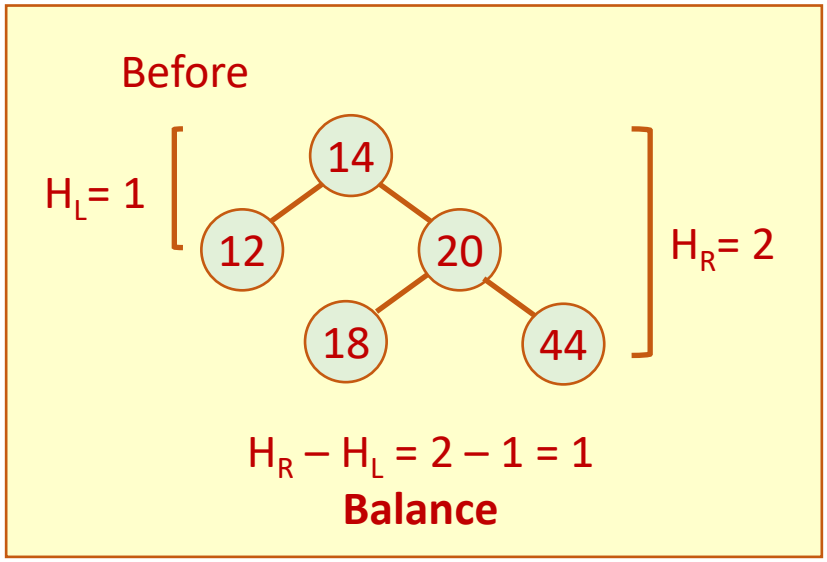
AVL Tree

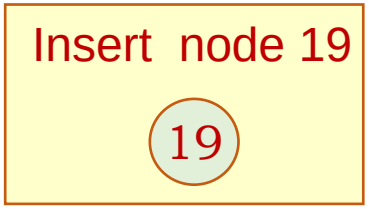

Before
18
12
20
4
14
H_L= 2
H_R=1
$H_L - H_R = 2 - 1 = 1$
Balance


After
18
12
20
4
14
13
H_L= 3
H_R=1
$H_L - H_R = 3 - 1 = 2$
Unbalance

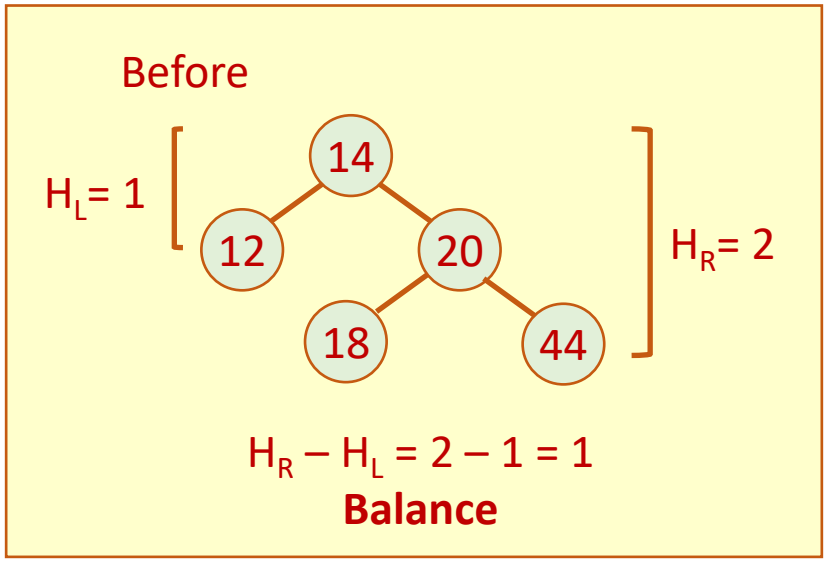

Insert node 13
13

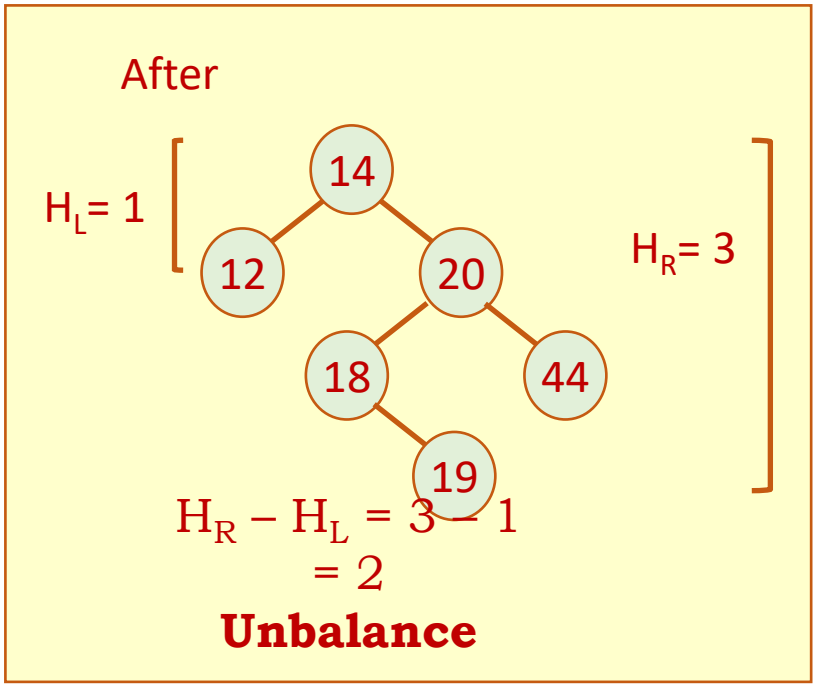
AVL Tree

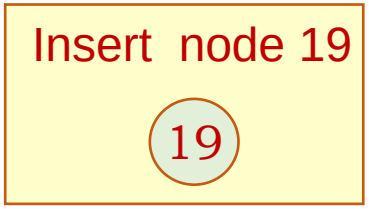

Before
H_L= 1
14
12
20
18
44
H_R= 2
$H_R - H_L = 2 - 1 = 1$
Balance


Insert node 19
19

AVL Tree


Before
H_L= 1
14
12
20
18
44
H_R= 2
$H_R - H_L = 2 - 1 = 1$
Balance


After
H_L= 1
14
12
20
18
44
19
H_R= 3
$H_R - H_L = 3 - 1$
$= 2$
Unbalance


Insert node 19
19

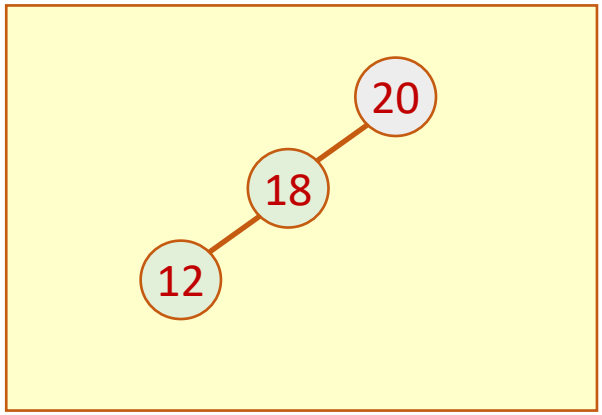
AVL Tree

Rotation

- When the tree structure changes during insertion or deletion, we need to transform the tree to restore the AVL tree property.
  - This is done using single rotations or double rotations

AVL Tree

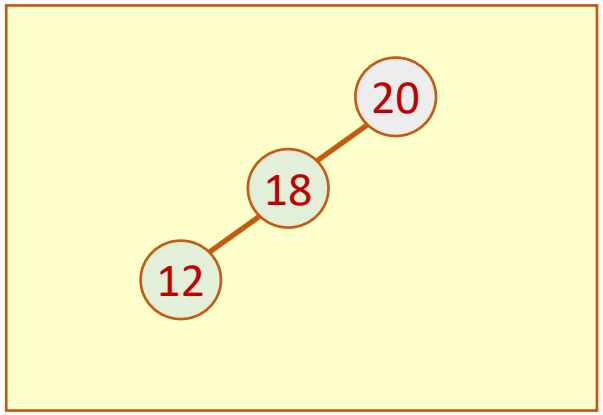
Single Rotation Right



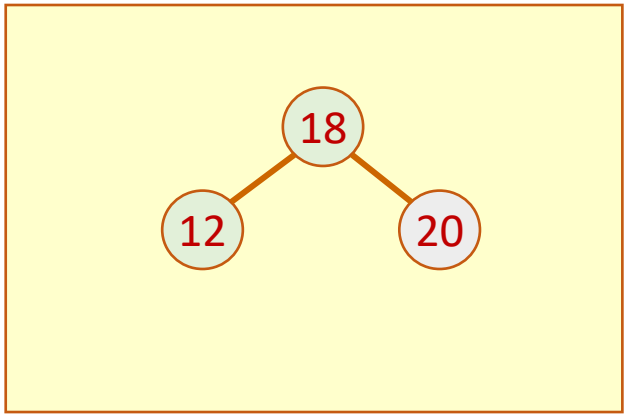
- Unbalanced
- To rotate right at node 20

AVL Tree

Single Rotation Right



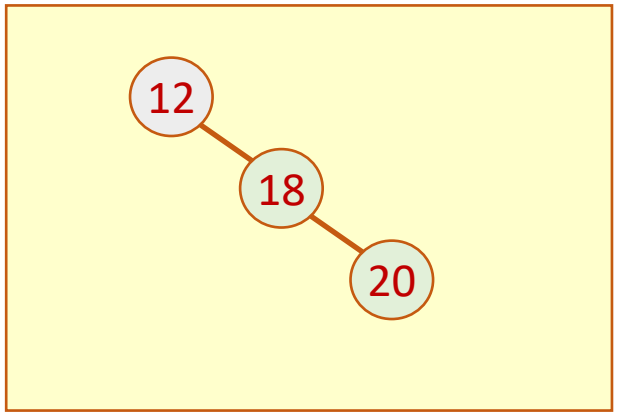
- Unbalanced
- To rotate right at node 20



- Balance the tree by rotating the root, 20, to the right so that it becomes the right sub-tree of 18.

AVL Tree

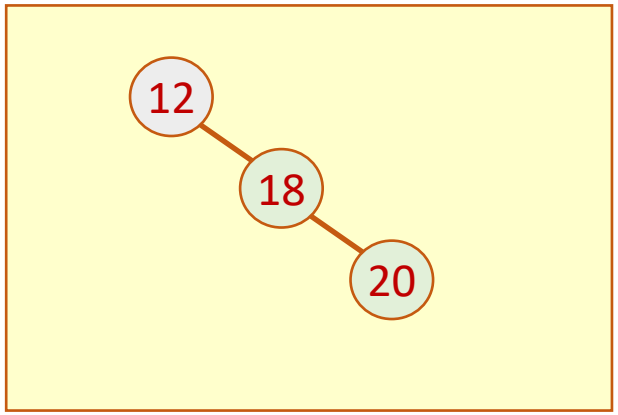
Single Rotation Left



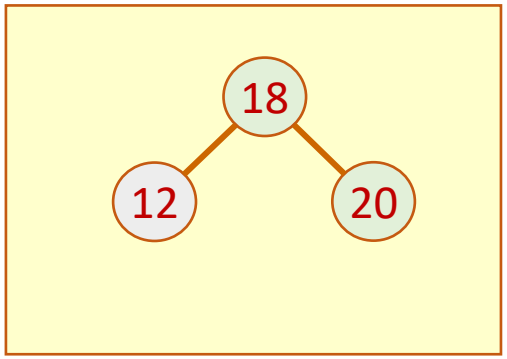
- Unbalanced
- To rotate left at node 12

AVL Tree

Single Rotation Left



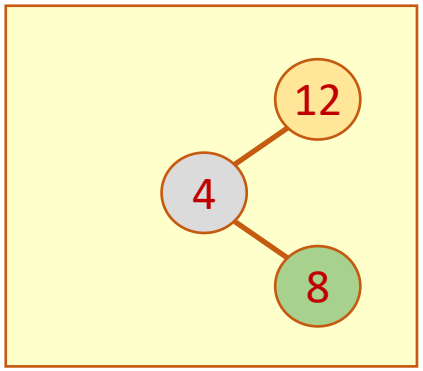
- Unbalanced
- To rotate left at node 12



- Balance the tree by rotating the root, 12, to the left so that it becomes the left sub-tree of 18.

AVL Tree

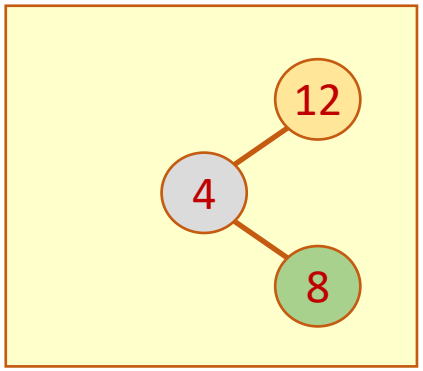
Double Rotation Left-Right



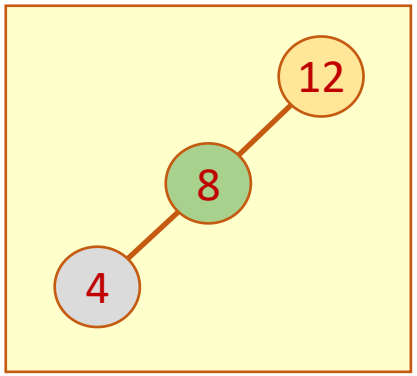
- Unbalanced
- To rotate left at node 4,
- Followed by a right rotation at node 12

AVL Tree

Double Rotation Left-Right



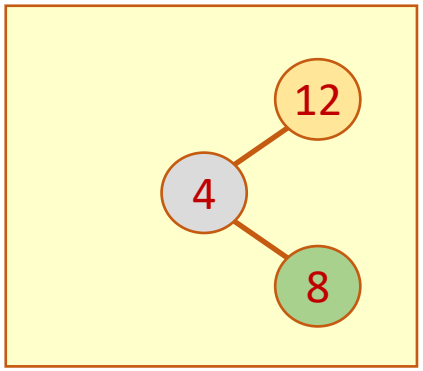
- Unbalanced
- To rotate left at node 4,
- Followed by a right rotation at node 12



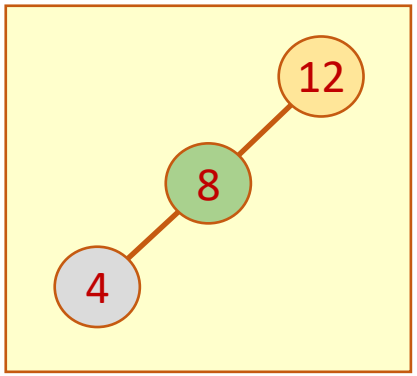
- Rotate the left sub-tree of node 12 at node 4 to the **left** so that the node 4 becomes the left sub-tree of 8.

AVL Tree

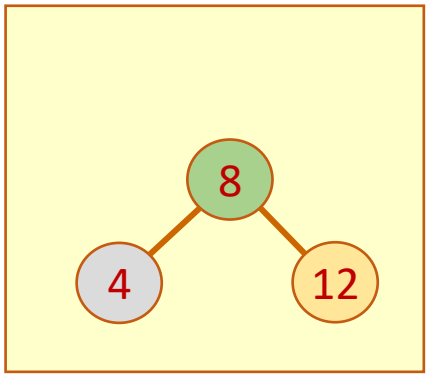
Double Rotation Left-Right



- Unbalanced
- To rotate left at node 4,
- Followed by a right rotation at node 12



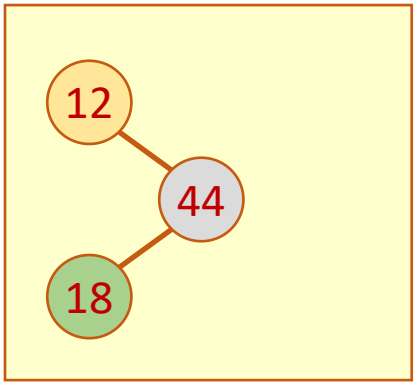
- Rotate the left sub-tree of node 12 at node 4 to the **left** so that the node 4 becomes the left sub-tree of 8.



- Balance the tree by rotating the root, 12, to the **right** so that it becomes the right sub-tree of 8.

AVL Tree

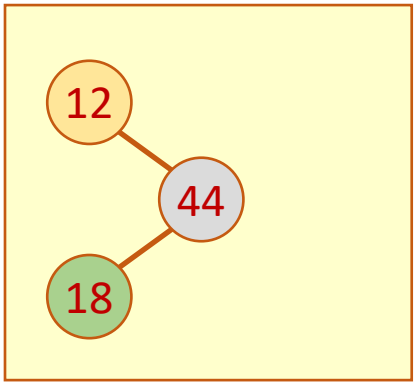
Double Rotation Right-Left



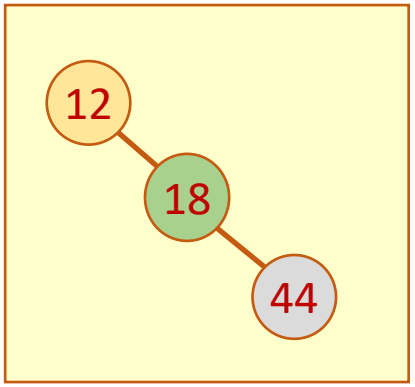
- Unbalanced
- To rotate right at node 44,
- Followed by a left rotation at node 12

AVL Tree

Double Rotation Right-Left



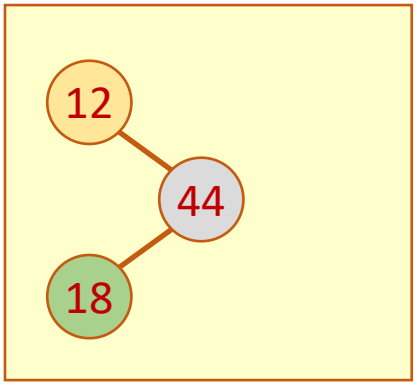
- Unbalanced
- To rotate right at node 44,
- Followed by a left rotation at node 12



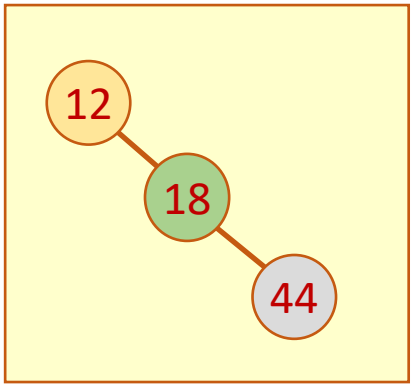
- **Right** rotate at node 44 so that node 44 becomes the right sub-tree of node 18

AVL Tree

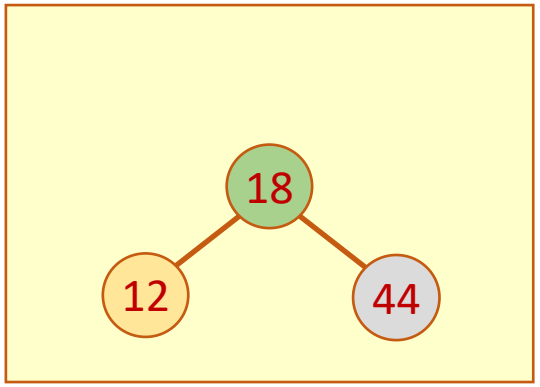
Double Rotation Right-Left



- Unbalanced
- To rotate right at node 44,
- Followed by a left rotation at node 12



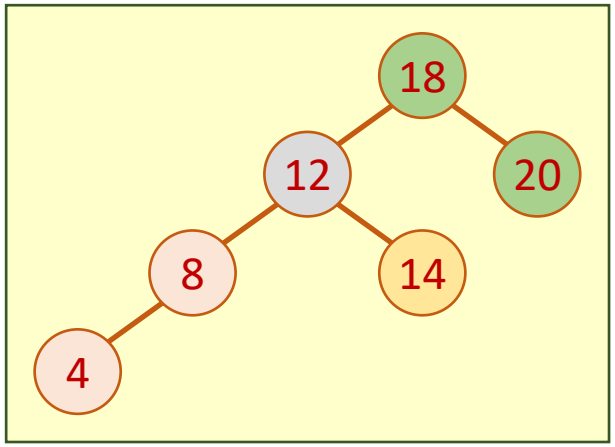
- **Right** rotate at node 44 so that node 44 becomes the right sub-tree of node 18



- **Left** rotate at node 12 so that node 12 becomes the left sub-tree of node 18

AVL Tree

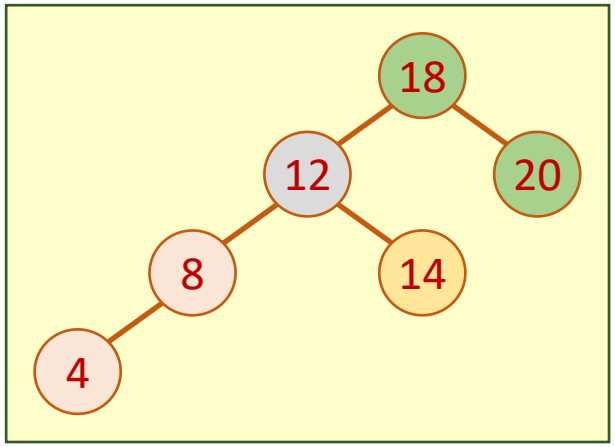
Complex Right Rotation

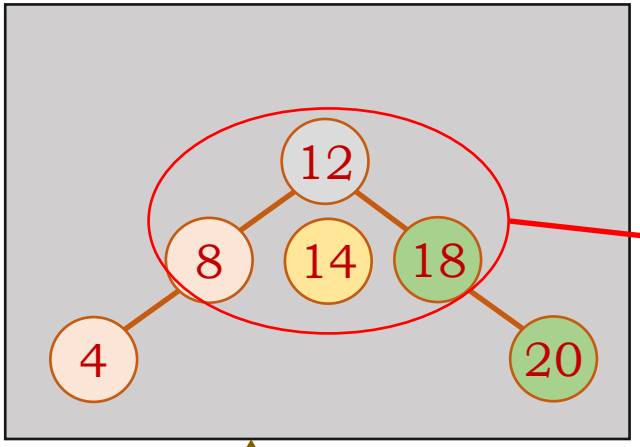


To preserve the property of the AVL tree, rotate right at node 18. Bringing node 12 up and detaching node 14; make node 18 the right sub-tree of node 12.

AVL Tree

Complex Right Rotation





This is not a property of an AVL Tree

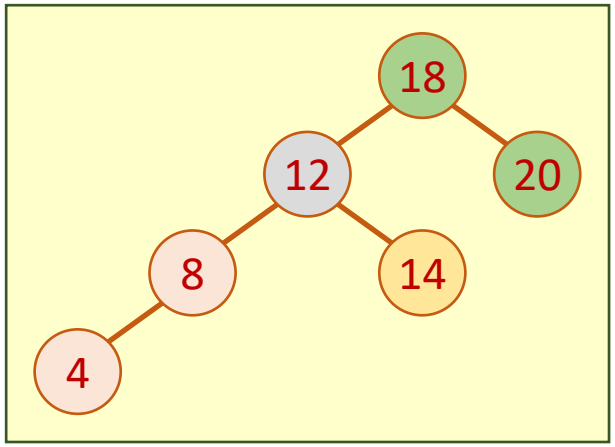
To preserve the property of the AVL tree, rotate right at node 18. Bringing node 12 up and detaching node 14; make node 18 the right sub-tree of node 12.

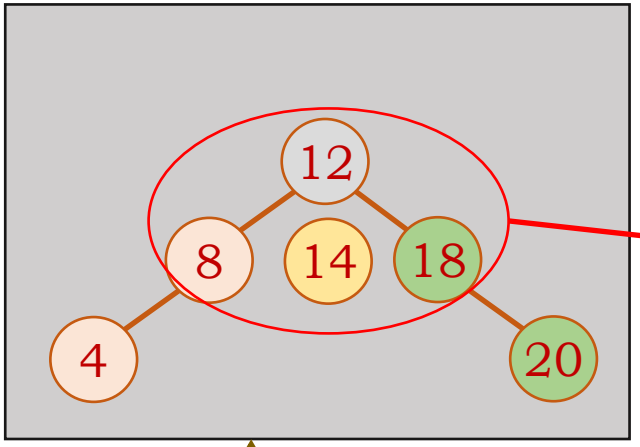
In the right rotation, the node 14 loses its parent.

Since the left sub-tree of node 18 is detached, it is used to attach node 14.

AVL Tree

Complex Right Rotation



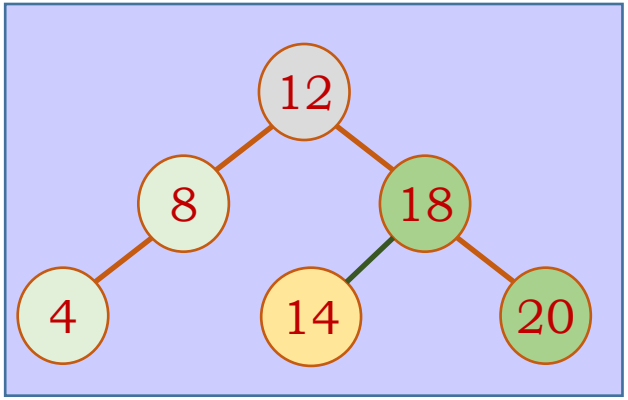


This is not a property of an AVL Tree

To preserve the property of the AVL tree, rotate right at node 18. Bringing node 12 up and detaching node 14; make node 18 the right sub-tree of node 12.

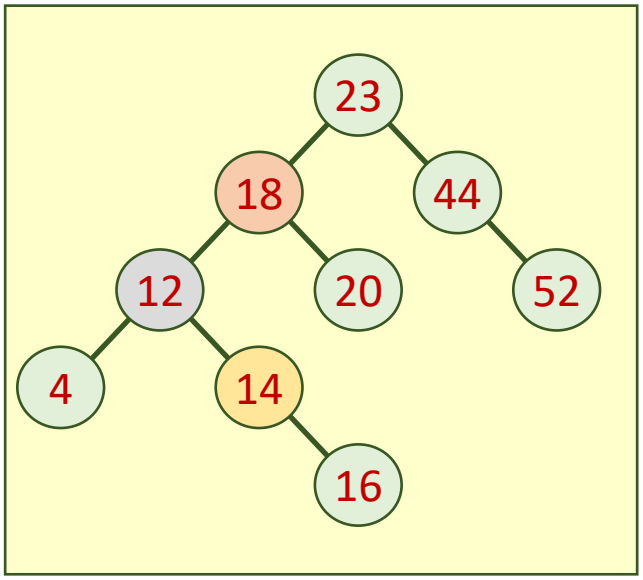
In the right rotation, the node 14 loses its parent.

Since the left sub-tree of node 18 is detached, it is used to attach node 14.



AVL Tree

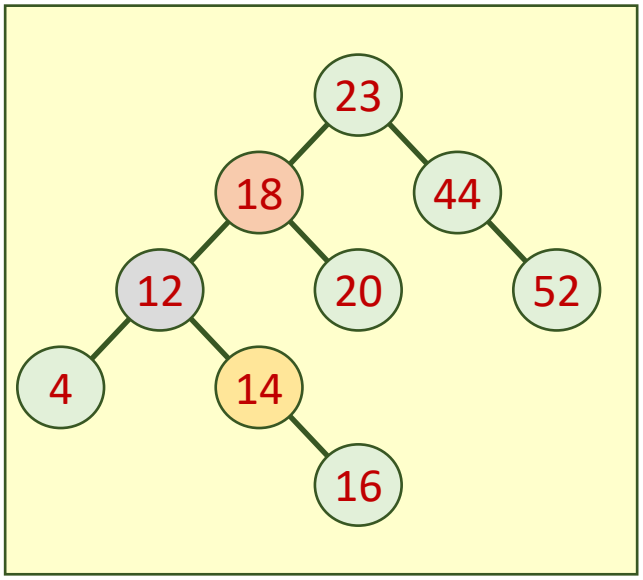
Complex Double Left-Right Rotation



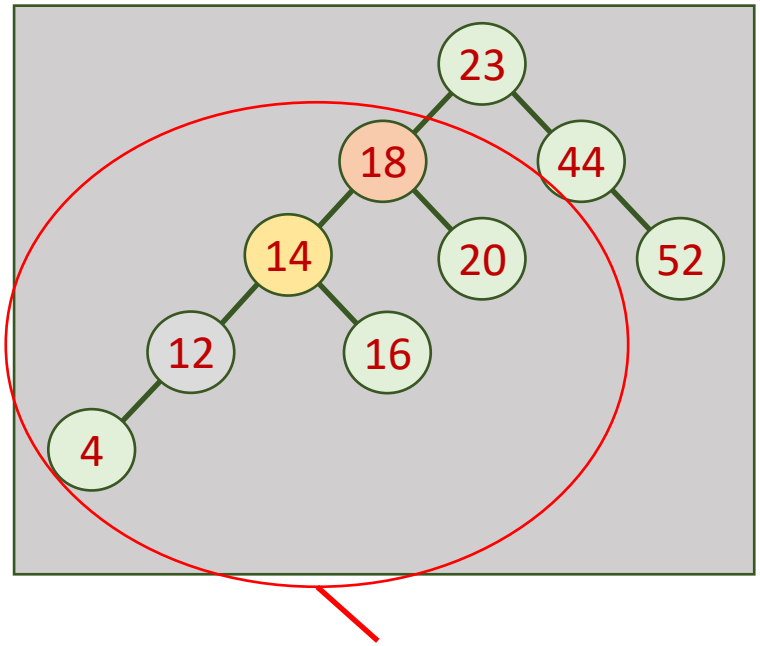
Left rotate at node 12 so that it becomes the left sub-tree of the node 14.

AVL Tree

Complex Double Left-Right Rotation



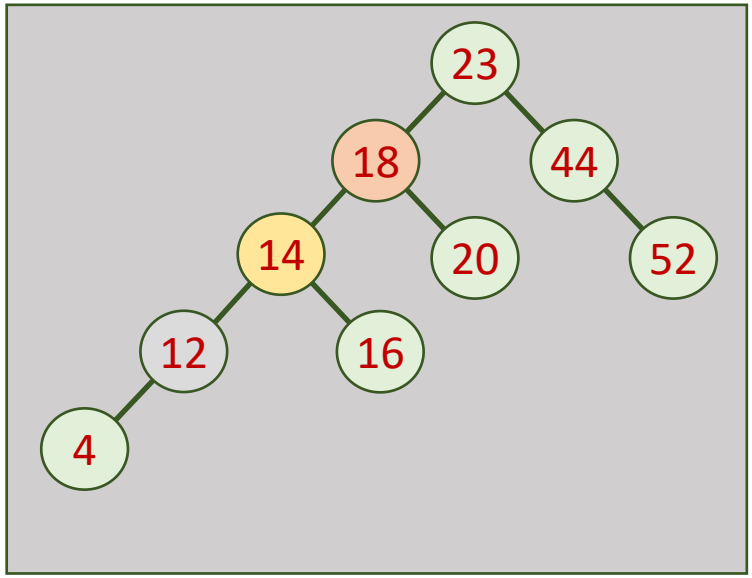
Left rotate at node 12 so that it becomes the left subtree of the node 14.



Tree is unbalance at node 18. Need further rotation.

AVL Tree

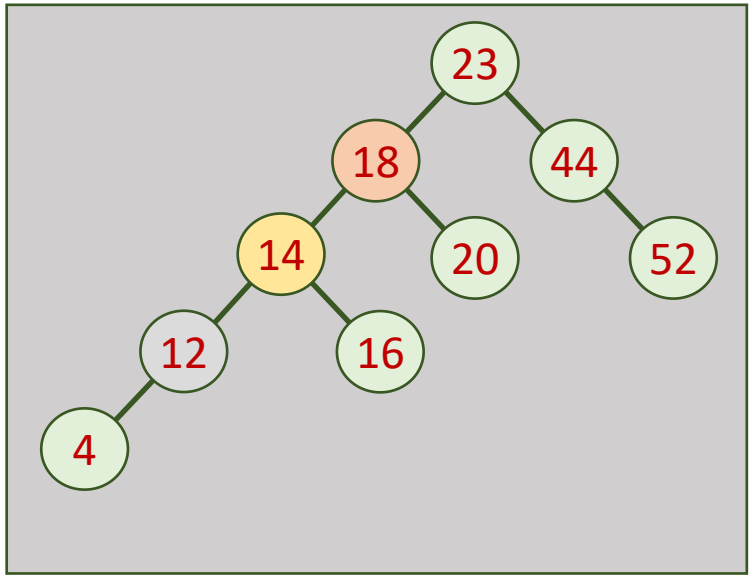
Complex Double Left-Right Rotation

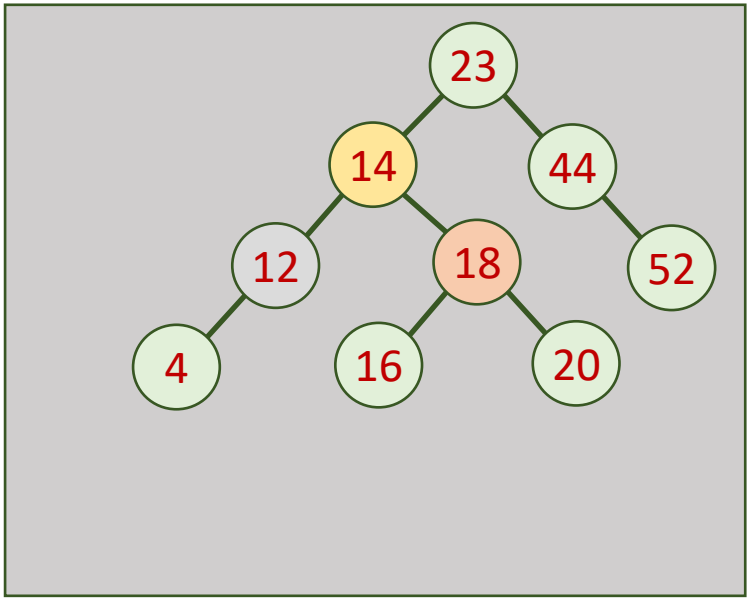


Right rotate at node 18 so that node 18 becomes the right sub-tree of the node 14 and the node 16 becomes the left sub-tree of node 18.

AVL Tree

Complex Double Left-Right Rotation

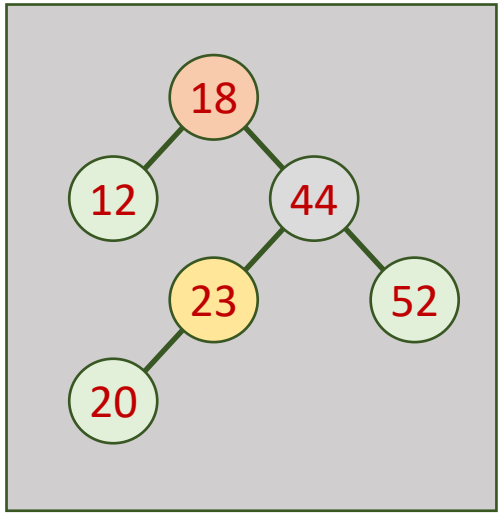




Right rotate at node 18 so that node 18 becomes the right sub-tree of the node 14 and the node 16 becomes the left sub-tree of node 18.

AVL Tree

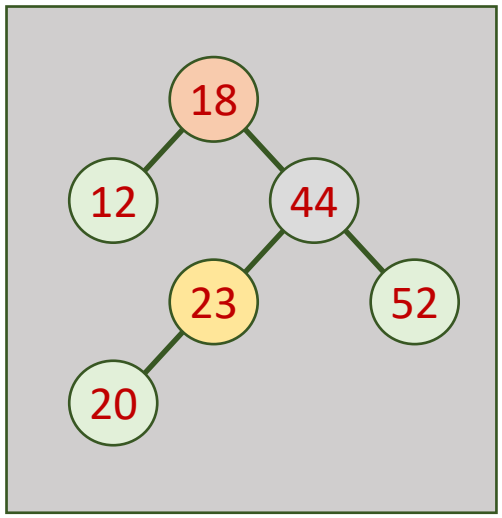
Complex Double Right-Left Rotation

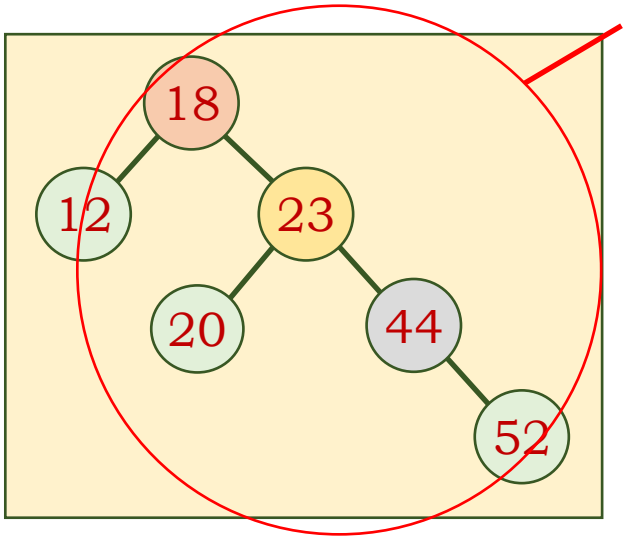


To balance the tree, right rotate at node 44 so that node 23 becomes the left sub-tree of node 44 and node 44 becomes the right sub-tree of node 18.

AVL Tree

Complex Double Right-Left Rotation



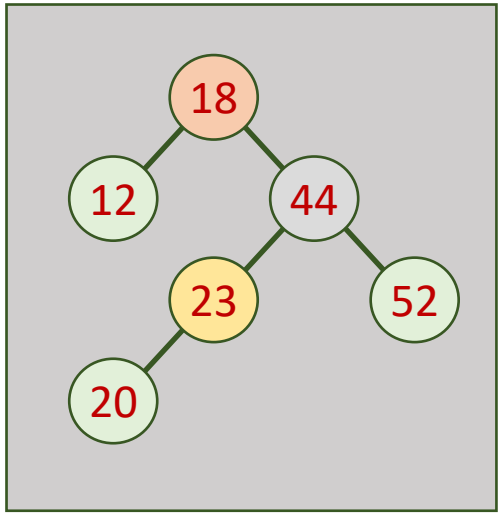


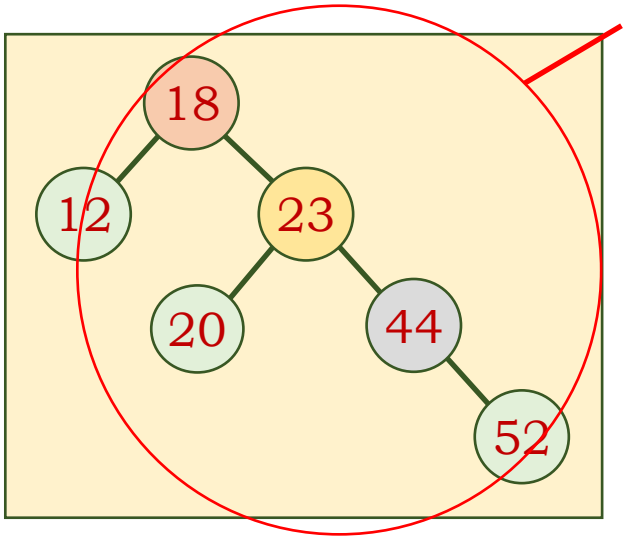
Tree is unbalance at node 18. Need further left rotation at node 23 to balance.

To balance the tree, right rotate at node 44 so that node 23 becomes the left sub-tree of node 44 and node 44 becomes the right sub-tree of node 18.

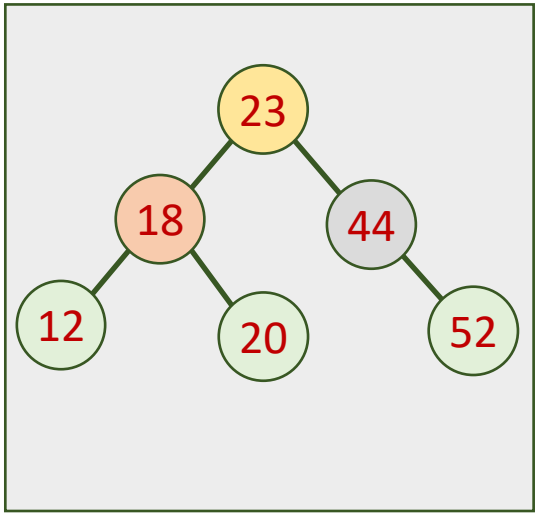
AVL Tree

Complex Double Right-Left Rotation





Tree is unbalance at node 18. Need further left rotation at node 23 to balance.



To balance the tree, right rotate at node 44 so that node 23 becomes the left sub-tree of node 44 and node 44 becomes the right sub-tree of node 18.

AVL Tree

Algorithm AVLInsert (ref root <tree pointer>,
val newPtr <tree pointer>,
ref taller <Boolean>)

Using recursion, insert a node into an AVL tree.

Pre: root is a pointer to first node in AVL tree/subtree
newPtr is a pointer to new node to be inserted

Post: taller is a Boolean: true indicating the subtree height has increased, false indicating same height

Return: root returned recursively up the tree.

AVL Tree

```
if (root null)
   // Insert at root
   root = newPtr
   taller = true
   return root
end if
```

AVL Tree

```
if (newPtr.key < root.key)
    root.left = AVLInsert (root.left, newPtr, taller)
    if (taller)
        // Left subtree is taller
        if (root left-high)
            root = leftBalance (root, taller)
        elseif (root even-high)
            root.bal = left-high
        else
            // Was right high but now even high
            root.bal = even-high
            taller = false
        end if
     end if
```

AVL Tree

```
else
    // new data >= root data
    root.right = AVLInsert (root.right, newPtr, taller)
    if (taller)
        // Right subtree is taller
        if (root left-high)
            root.bal = even-high
            taller = false
        elseif (root even-high)
            //Was balanced but now right high
            root.bal = right-high
        else
            root = rightBalance (root, taller)
        end if
    end if
end if
return root
End AVLInsert
```

AVL Tree

Algorithm rotateRight (ref root <tree pointer>)

This algrithm exchanges pointer to rotate the tree right.

Pre: root points to tree to be rotated

Post: node rotated and root updated

tempPtr = root.left

root.left = tempPtr.right

tempPtr.right = root

return tempPtr

End rotateRight

AVL Tree

Algorithm rotateLeft (ref root <tree pointer>)

This algorithm exchanges pointers to rotate the tree left.

Pre: root pointes to tree to be rotated

Post: node rotated and root updated

tempPtr = root.right

root.right = tempPtr.left

tempPtr.left = root

return tempPtr

End rotateLeft

AVL Tree

Algorithm doubleLeftRightRotate (ref root <tree pointer>)

This algorithm exchanges pointers to double rotate the tree left followed by right.

Pre: root pointes to tree to be rotated

Post: node rotated and root updated

tempPtr = leftRotate (root.left)

root.left = tempPtr

tempPtr = rightRotate(root)

return tempPtr

End doubleLeftRightRotate

AVL Tree

Algorithm doubleRightLeftRotate (ref root <tree pointer>)

This algorithm exchanges pointers to double rotate the tree right followed by left.

Pre: root pointes to tree to be rotated

Post: node rotated and root updated

tempPtr = rightRotate (root.right)

root.right = tempPtr

tempPtr = leftRotate(root)

return tempPtr

End doubleRightLeftRotate

AVL Tree

- The AVL delete follows the basic logic of the binary search tree (BST) delete with the addition of the logic to balance the tree.
- The balancing algorithms are:
  - Delete Right Balance
  - Delete Left Balance (this algorithm mirror the Delete Right Balance)

AVL Tree

- The balancing algorithm is required only if after the deletion, the AVL tree becomes unbalance.
- For example, we deleted on the left of a tree or sub-tree that was already right high, the tree or sub-tree would now becomes doubly right high. In this case, we need to rotate the right sub-tree to the left.

AVL Tree

- If the right sub-tree is left high, we need to rotate twice, first to the right and then to the left. We then adjust the balance factor of the tree or sub-tree accordingly.
- If the right sub-tree is NOT left high, we need to rotate only once. Before the rotation, however, we adjust the balance factors accordingly.

AVL Tree

Adjusting Balance Factor

- After an insert or delete, as we balance the tree, we must adjust the balance factors. Generally:
  - If the root was even balanced before an insert, it is now high on the side in which the insert was made
  - If an insert was in the shorter sub-tree of a tree that was not even balanced, the root is now even balanced
  - If an insert was in the higher sub-tree of a tree that was not even balanced, the root must be rotated.

AVL Tree

- Deletion from AVL-Trees
  - Unlike insertion, deletion can seriously unbalance AVL-Trees
    - A single (or double) rotation may not fix it up
  - We can often get away with a cheat
    - “Lazy deletion”
    - Don’t delete the node, just flag it
    - Re-use the node if we can at a later insertion


Running time complexity of AVL Tree
What is the running time complexity of AVL tree?

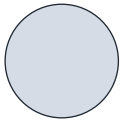
AVL Tree

The height of an AVL tree with n nodes has height h upper bounded by $O(\lg n)$.

Proof:

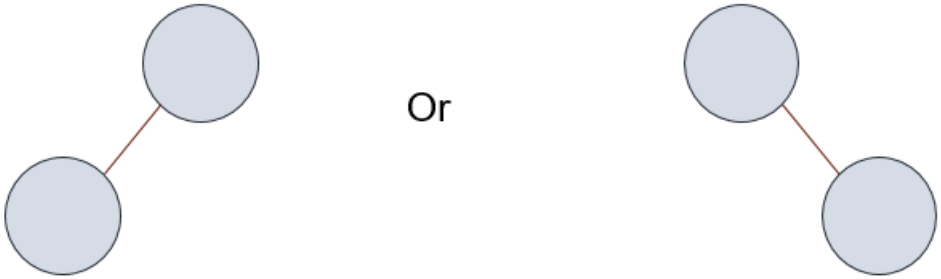
$Let\ n_h$ be the minimum number of nodes in AVL tree with height h. Hence, for height $0, n_0 = 1$, and height $1, n_1 = 2$. That is,

For height 0, $n_0 = 1$:



AVL Tree

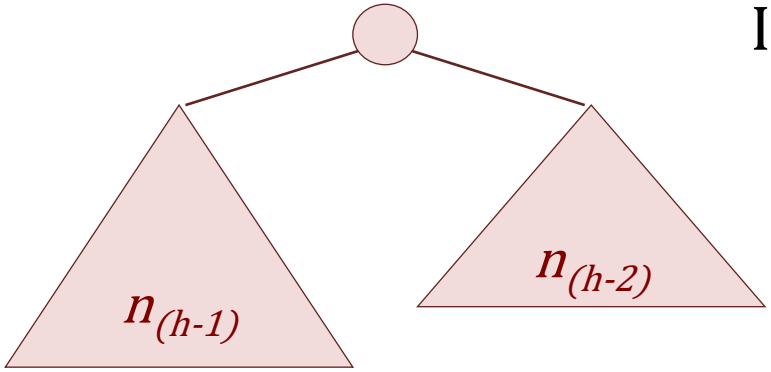
For height 1, $n_1 = 2$:



How about when $h \geq 2$?

AVL Tree

How about when $h \geq 2$?

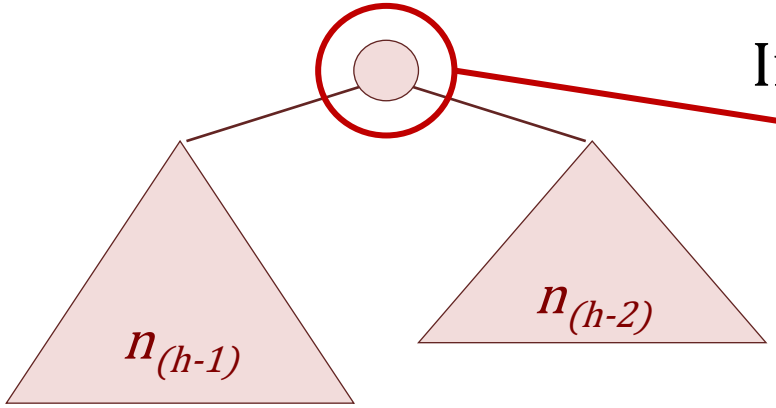


In general, for $h \geq 2$,

$n_h =$

AVL Tree

How about when $h \geq 2$?

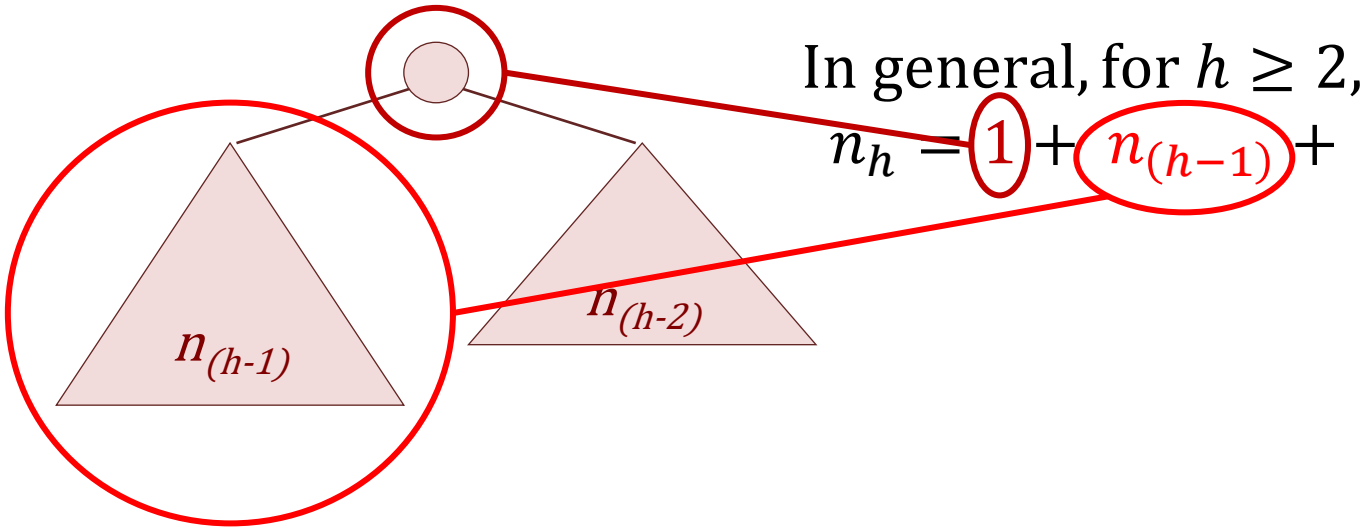


In general, for $h \geq 2$,

$n_h = 1 +$

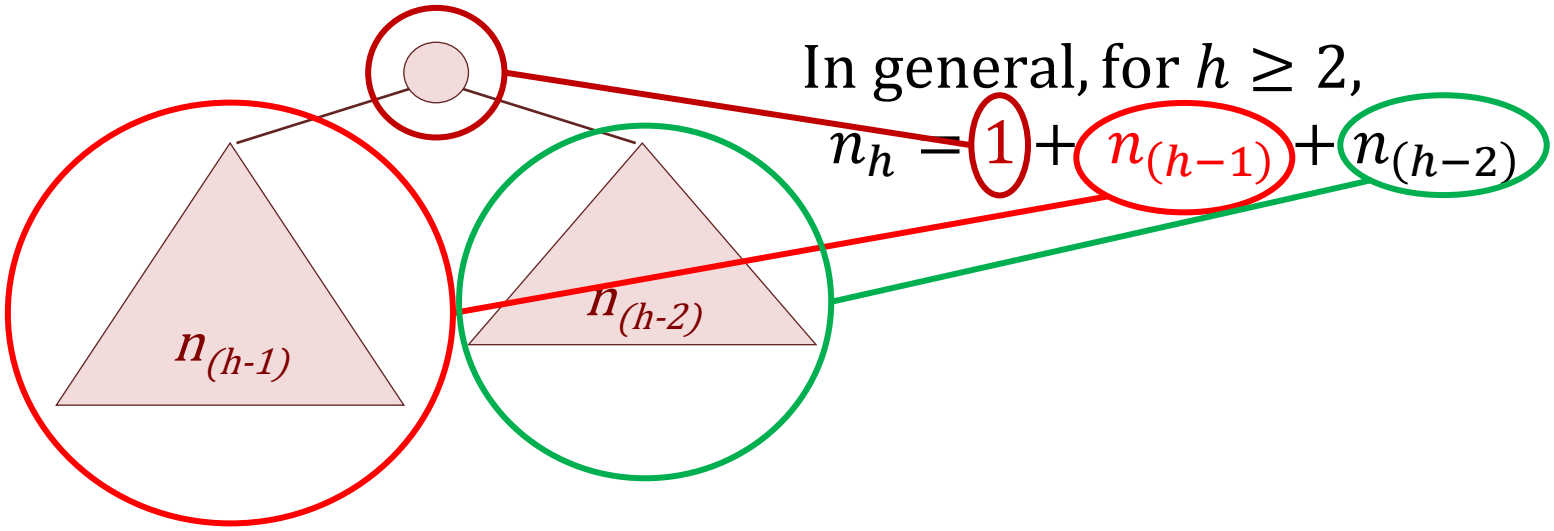
AVL Tree

How about when $h \geq 2$?



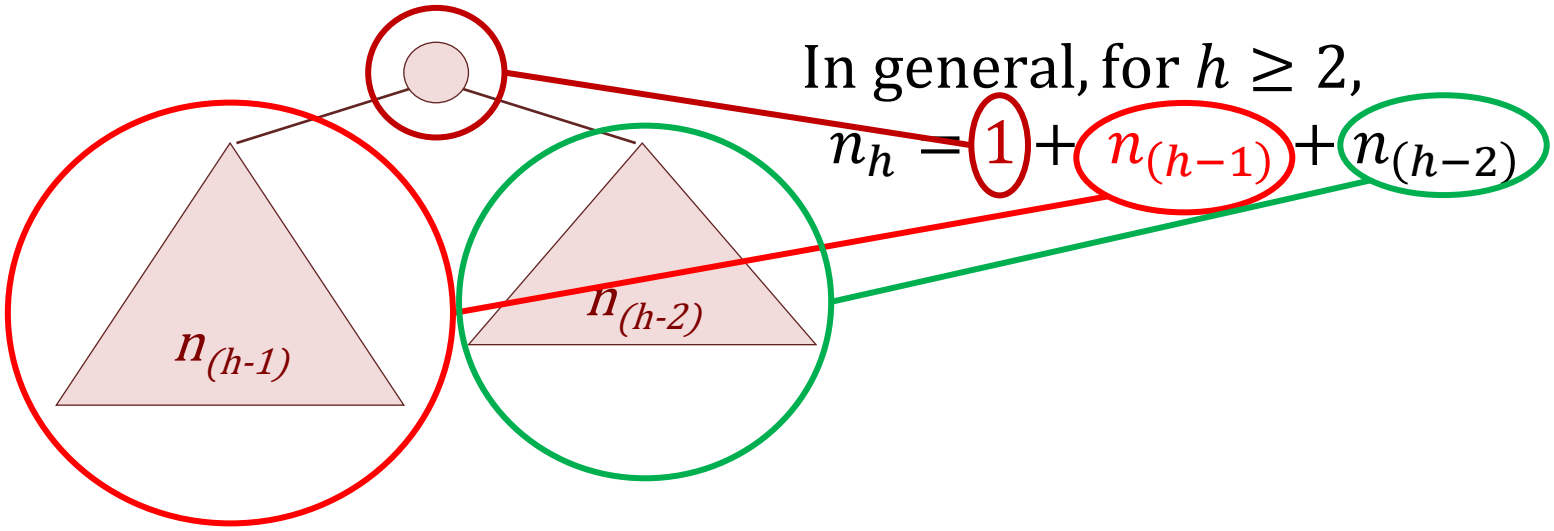
AVL Tree

How about when $h \geq 2$?



AVL Tree

How about when $h \geq 2$?



$$Hence, n_h = 1 + n_{(h-1)} + n_{(n-2)}$$

AVL Tree

- Next, we need to prove that for the minimum number of nodes $n_{(h)}$, the height of the AVL tree is upper-bounded by $O(\log_2 n)$.

  Height of AVL tree: $n_h = 1 + n_{(h-1)} + n_{(h-2)}$

  First, we will look at the left sub-tree:

  $n_{(h-1)} = 1 + n_{(h-2)} + n_{(h-3)}$

  Substituting $n_{(h-1)}$ $into$ $n_{(h)}$, we have

  $$n_{(h)} = 1 + \left(1 + n_{(h-2)} + n_{(h-3)}\right) + n_{(h-2)}$$
  $$= 2 + 2n_{(h-2)} + n_{(h-3)}$$

AVL Tree

Since the minimum number of nodes are strictly increasing as height of the tree increases, that is, $n_{(1)} < n_{(2)} < n_{(3),\ldots}$ etc. Hence we can say (generalize) that $n_{(h)} > 2n_{(h-2)}$.

Now we recursively expanding the expression…

$n_{(h)} > 2n_{(h-2)}$, we have

$n_{(h)} > 2\left(2n_{((h-2)-2)}\right)$

$n_{(h)} > 2\left(2\left(2n_{(((h-2)-2)-2)}\right)\right)$

…

$n_{(h)} > 2^k n_{((h-2)-2(k-1))}$

Note:

$2\left(2\left(2n_{(((h-2)-2)-2)}\right)\right)$

$2^3 n_{((h-2)-4)}$

AVL Tree

Note: $(h-2)-2(k-1)=0$
$h-2-2k+2=0$
$h-2k=0$
$h=2k \quad \therefore k = {}^{h}/_{2}$

The expansion continues until $(h-2)-2(k-1)=0$, that is, the root level. Rearranging the expression, we have, $k=\frac{h}{2}$.

Substituting k to $n_{(h)} > 2^k n_{((h-2)-2(k-1))}$, we get

$$n_{(h)} > 2^{\frac{h}{2}} n_{\left((h-2)-2\left(\frac{h}{2}-1\right)\right)}$$

Note: $(h-2)-2\left(\frac{h}{2}-1\right)$
$=(h-2)-(h-2)$
$=0$

$$n_{(h)} > 2^{\frac{h}{2}} n_{(0)}$$

$n_{(h)} > 2^{\frac{h}{2}}$ Since the number of node at $n_{(0)}=1$.

AVL Tree

Taking logarithmic base 2 on both sides of the equation, we have

$$\lg n_{(h)} > \lg 2^{\frac{h}{2}}$$

$$\lg n_{(h)} > \frac{h}{2} \qquad \text{Note: } \lg 2^x = x$$

$$h < 2 \lg n_{(h)}$$

Hence, $h \in O(\lg n)$

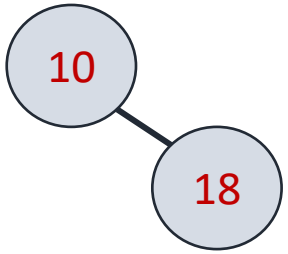
Trinode Restructuring of AVL tree

Trinode restructuring

- Trinode restructuring is a technique used to balance (restructure) an AVL tree. It looks at three nodes (the root of sub-trees) involved in the new imbalance of an AVL tree and re-arranges these three nodes to restore the balance property.

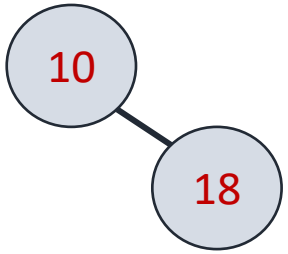
Trinode Restructuring method

We will see how trinode restructuring method works by example with an initial AVL tree as follow:

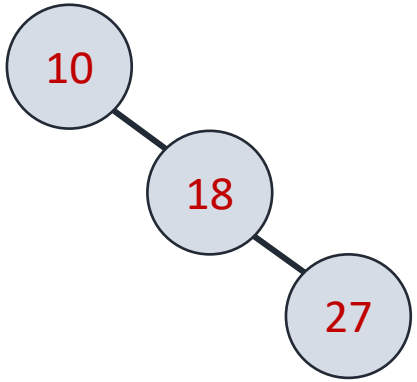


Trinode Restructuring method

We will see how trinode restructuring method works by example with an initial AVL tree as follow:



Insert a new node 27. With the new insertion, the AVL tree will be as follow:

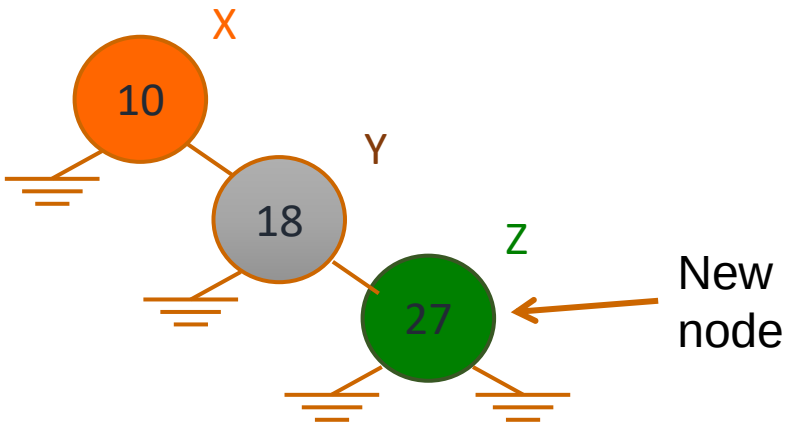


The AVL tree becomes unbalance.

Need to rebalance the tree!

Trinode Restructuring method

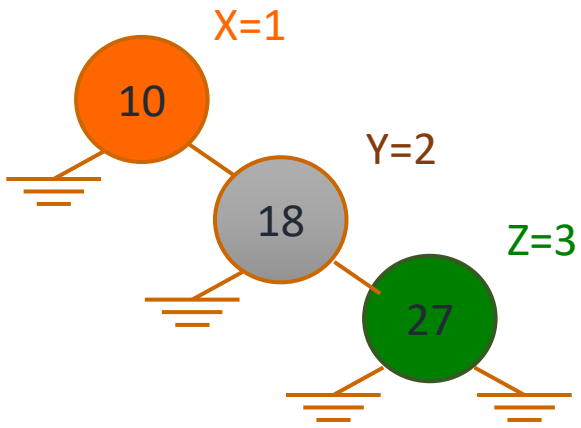
- To rebalance, after inserting a new node, follow the path up towards the root from it (new node) until we find an unbalanced node – call this node x. In this example, it is node with value 10.
- Call the higher-height child of x y (in this example, it is the node with value 18), and call the higher-height child of y z (in this example, it is the node with value 27).



Trinode Restructuring method

Execute the following:

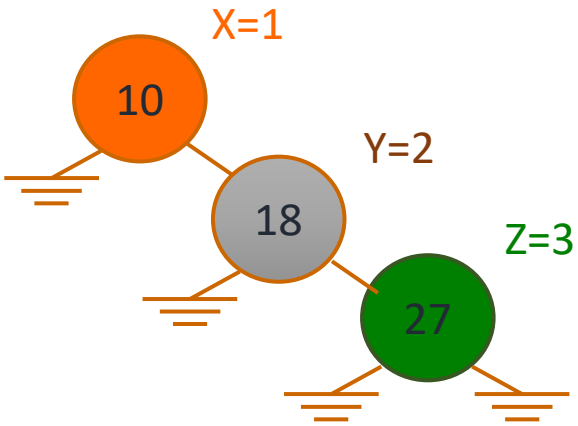
- Traverse the three nodes, X, Y, and Z in in-order and assign the nodes with a number 1, 2, and 3 where 1, 2, and 3 is the sequence number where the nodes is visited in **in-order**.



Trinode Restructuring method

Execute the following:

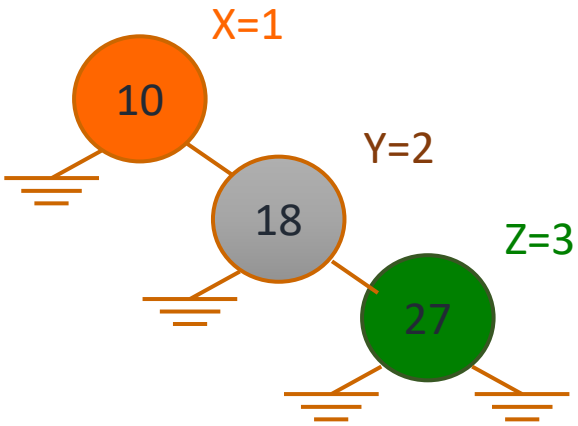
- Once identified, perform the required rotation(s) such that the node with value 2 will be the root of a sub-tree, the node with value 1 will be the root of the left-sub-tree, and the node with value 3 will be the root of the right-sub-tree.



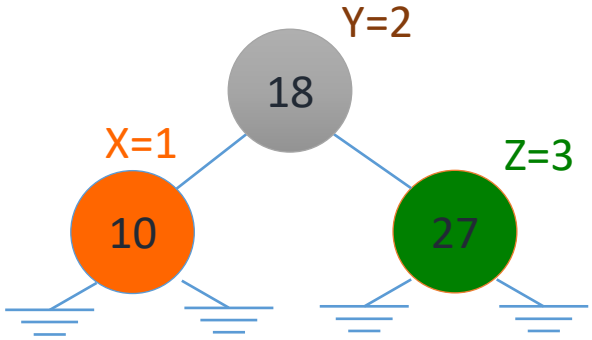
Trinode Restructuring method

Execute the following:

- Once identified, perform the required rotation(s) such that the node with value 2 will be the root of a sub-tree, the node with value 1 will be the root of the left-sub-tree, and the node with value 3 will be the root of the right-sub-tree.

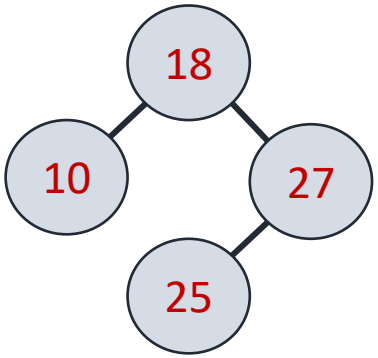


Perform a single left-rotate at node 10, and we have →

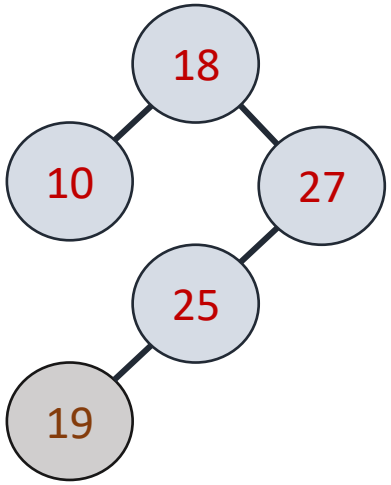


Trinode Restructuring method

Another example:

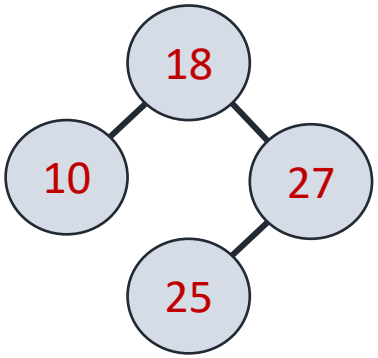


Insert 19 to the AVL tree, and we have:

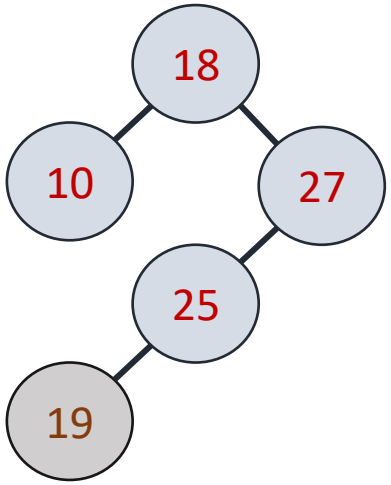


Trinode Restructuring method

Another example:



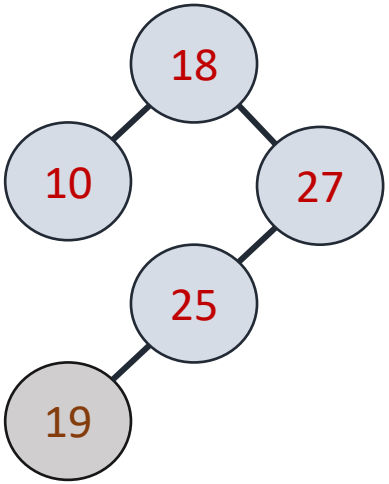
Insert 19 to the AVL tree, and we have:



The AVL tree becomes unbalance.

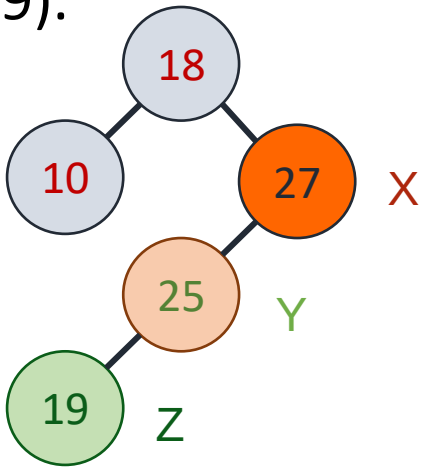
Need to rebalance the tree!

Trinode Restructuring method

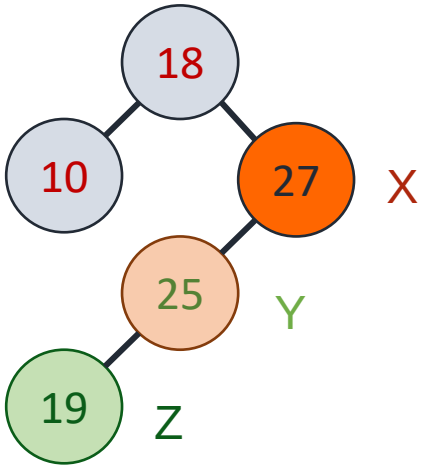


To rebalance, follow the path up towards the root from it (new node) until we find an unbalanced node – call this node x. In this example, it is node with value 27.

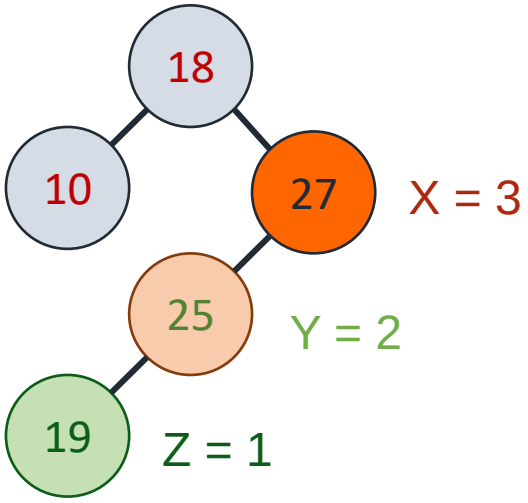
Call the higher-height child of x y (in this example, it is the node with value 25), and call the higher-height child of y z (in this example, it is the node with value 19).



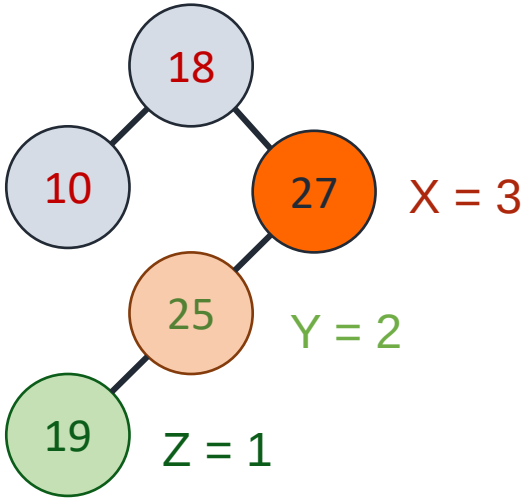
Trinode Restructuring method



Next, traverse the three nodes, X, Y, and Z in in-order and assign the nodes with a number 1, 2, and 3 where 1, 2, and 3 is the sequence number where the nodes is visited in **in-order.**

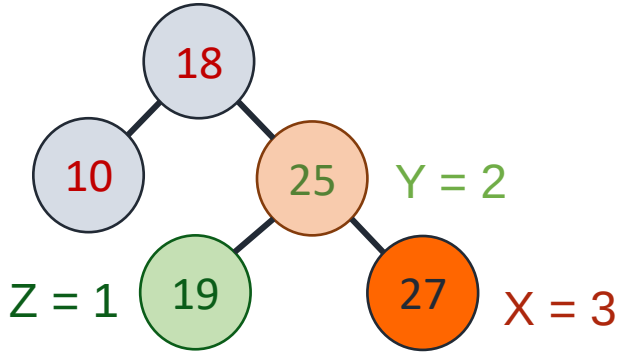


Trinode Restructuring method



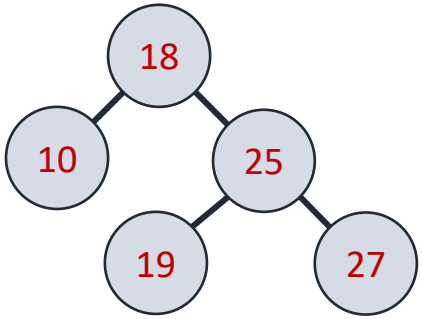
Once identified, perform the required rotation(s) such that the node with value 2 will be the root of a sub-tree, the node with value 1 will be the root of the left-sub-tree, and the node with value 3 will be the root of the right-sub-tree.

Perform a single right-rotation at node 27, and we have →

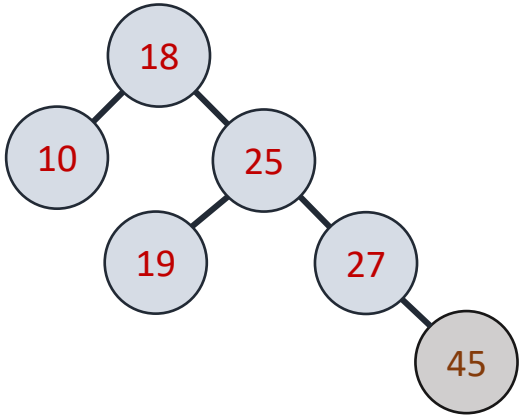


Trinode Restructuring method

Yet another example:

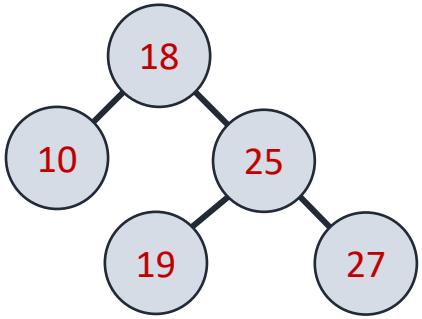


Insert 45 to the AVL tree, and we have:

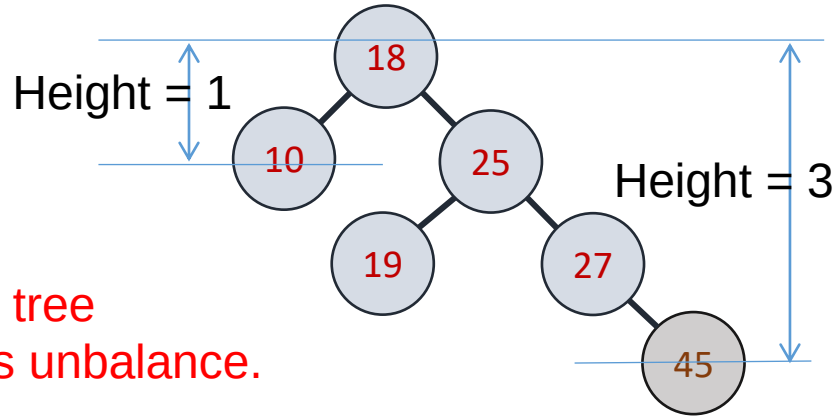


Trinode Restructuring method

Yet another example:



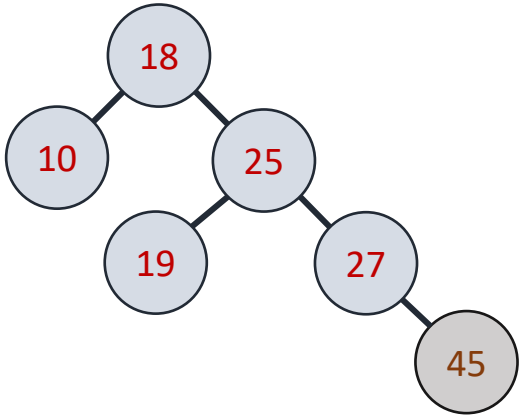
Insert 45 to the AVL tree, and we have:



The AVL tree becomes unbalance.

Need to rebalance the tree!

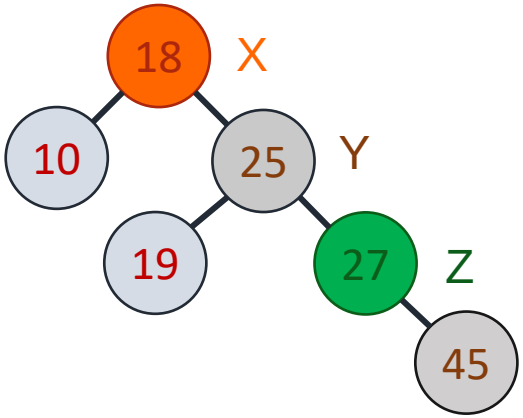
Trinode Restructuring method



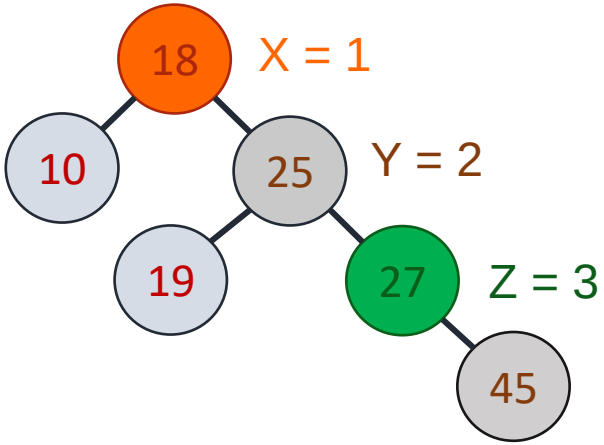
Follow the path up towards the root from it (new node) until we find an unbalanced node – call this node x. In this example, it is node with value 18.

Call the higher-height child of x y (in this example, it is the node with value 25), and call the higher-height child of y z (in this example, it is the node with value 27).

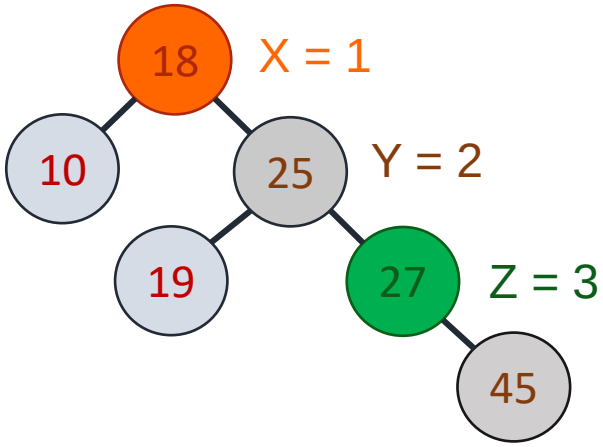
Trinode Restructuring method



Next, traverse the three nodes, X, Y, and Z in in-order and assign the nodes with a number 1, 2, and 3 where 1, 2, and 3 is the sequence number where the nodes is visited in **in-order.**

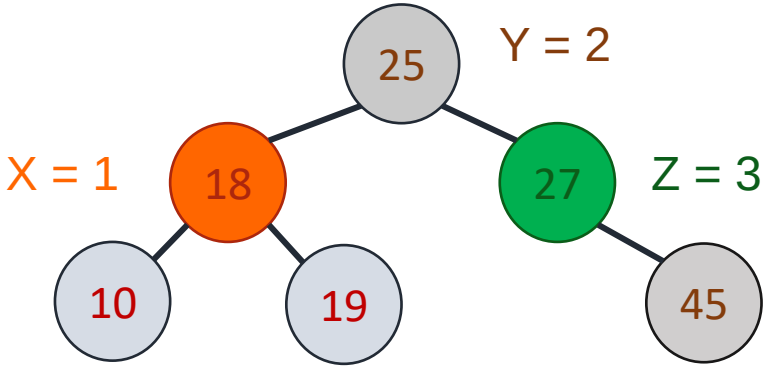


Trinode Restructuring method



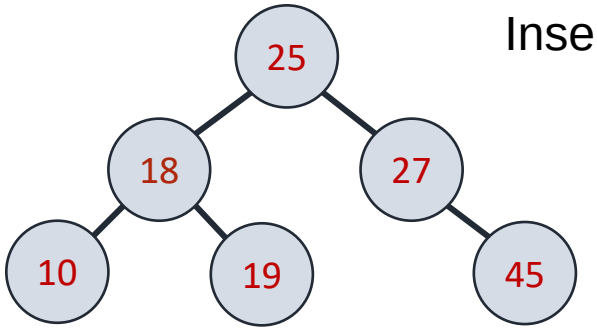
Once identified, perform the required rotation(s) such that the node with value 2 will be the root of a sub-tree, the node with value 1 will be the root of the left-sub-tree, and the node with value 3 will be the root of the right-sub-tree.

Perform a single complex left-rotation at node 18, and we have →

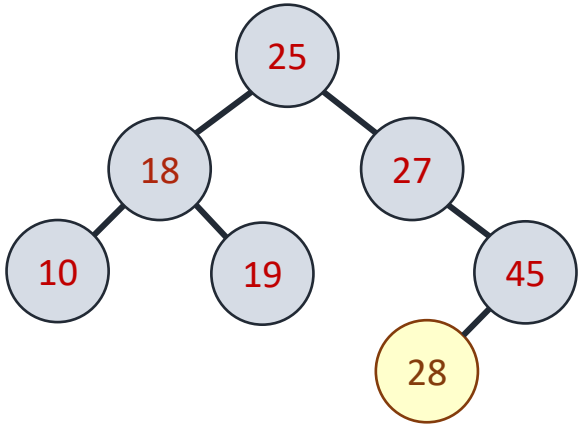


Trinode Restructuring method

Yet another example:

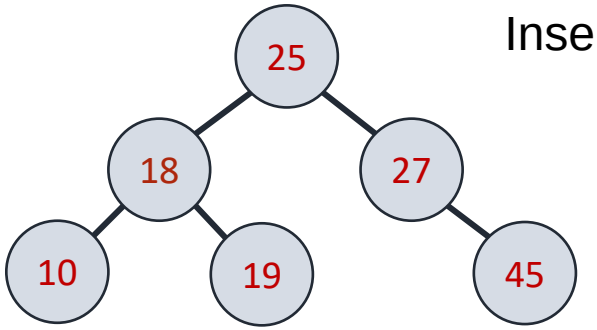


Insert 28 to the AVL tree, and we have:

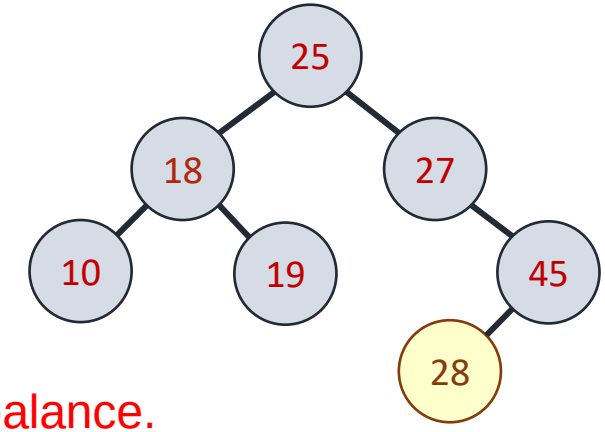


Trinode Restructuring method

Yet another example:



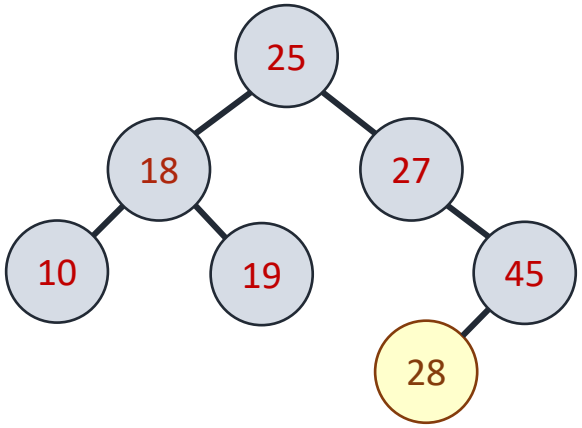
Insert 28 to the AVL tree, and we have:



The AVL tree becomes unbalance.

Need to rebalance the tree!

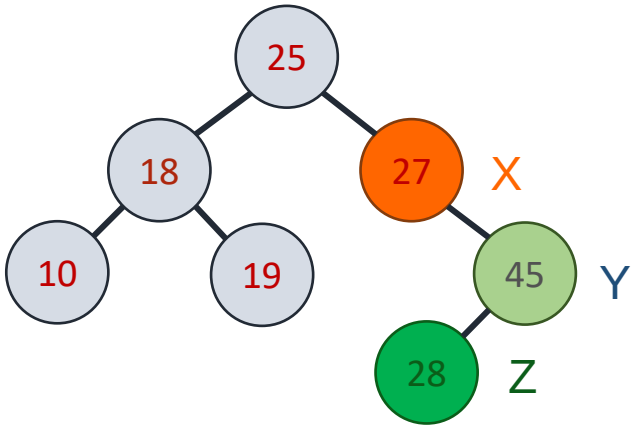
Trinode Restructuring method



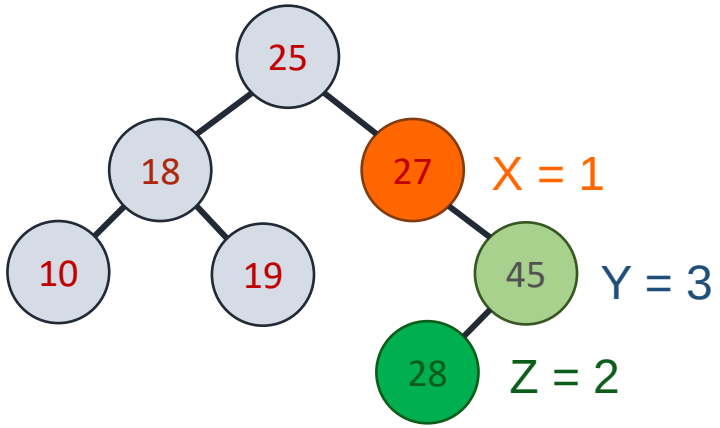
Follow the path up towards the root from it (new node) until we find an unbalanced node – call this node x. In this example, it is node with value 27.

Call the higher-height child of x y (in this example, it is the node with value 45), and call the higher-height child of y z (in this example, it is the node with value 28).

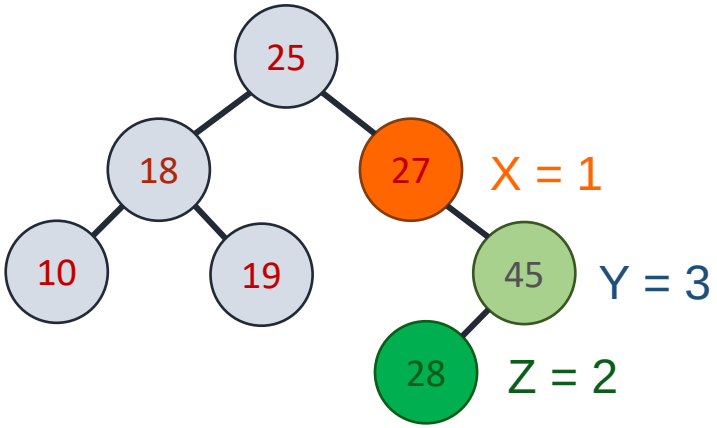
Trinode Restructuring method



Next, traverse the three nodes, X, Y, and Z in in-order and assign the nodes with a number 1, 2, and 3 where 1, 2, and 3 is the sequence number where the nodes is visited in **in-order.**



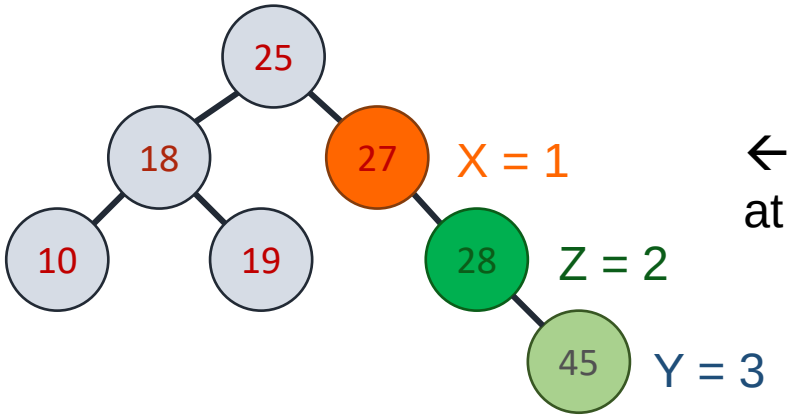
Trinode Restructuring method



Once identified, perform the required rotation(s) such that the node with value 2 will be the root of a sub-tree, the node with value 1 will be the root of the left-sub-tree, and the node with value 3 will be the root of the right-sub-tree.

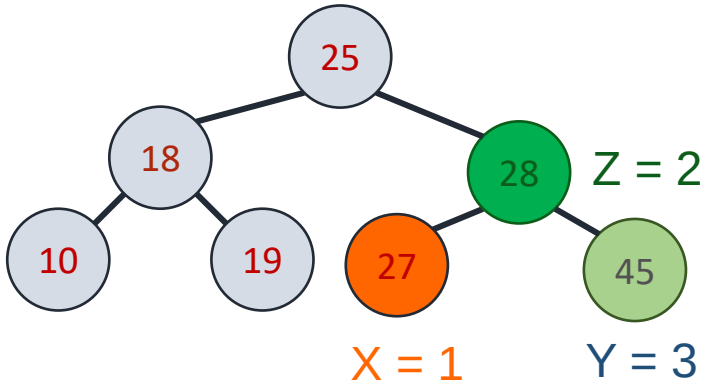
To correct the height of the AVL tree, we need to perform a double-right-left rotation by first **rotate right** at node 45 follows with a **left rotation** at node 27.

Trinode Restructuring method



← After single right-rotation at node 45.

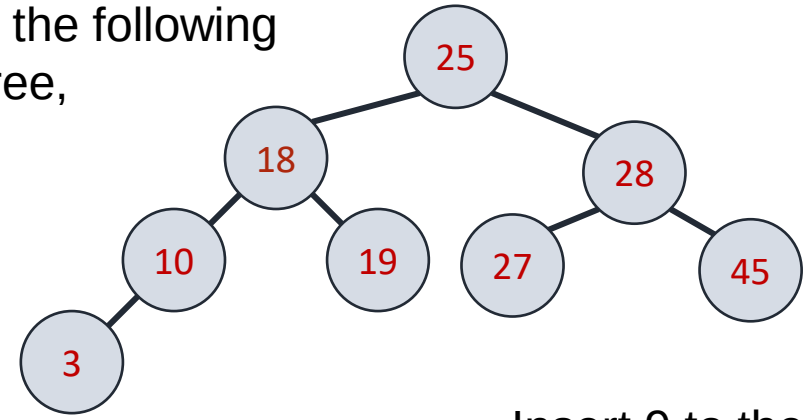
After single left-rotation at node 27 →.



Trinode Restructuring method

Take the following example as self exercise. Try it yourself. If you need help, check with me during tutorial session.

Given the following AVL tree,



Insert 9 to the AVL tree.

Construction of AVL Tree

Data Structure - AVL

Starting with an empty AVL tree, insert elements into the tree with the following keys, in that order, into an AVL tree **showing each step and any rotations performed**: 17, 40, 50, 49, 43, 52, 47, 45

17, 40, 50, 49, 43, 52, 47, 45

Insert 17

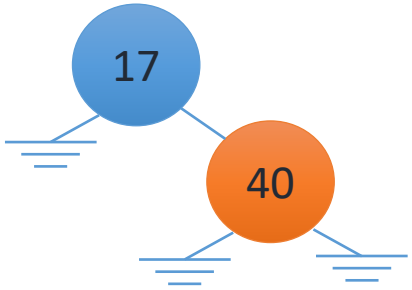
Data Structure - AVL

Starting with an empty AVL tree, insert elements into the tree with the following keys, in that order, into an AVL tree **showing each step and any rotations performed**: 17, 40, 50, 49, 43, 52, 47, 45

17, 40, 50, 49, 43, 52, 47, 45

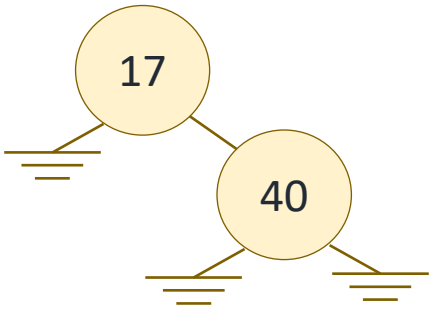
Insert 17

Insert 40



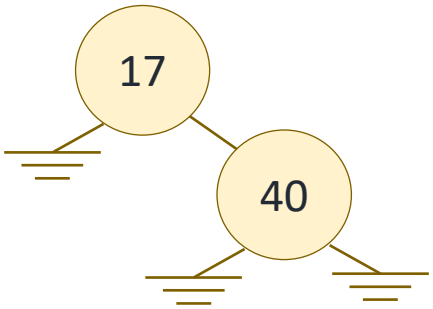
Data Structure - AVL

17, 40, 50, 49, 43, 52, 47, 45

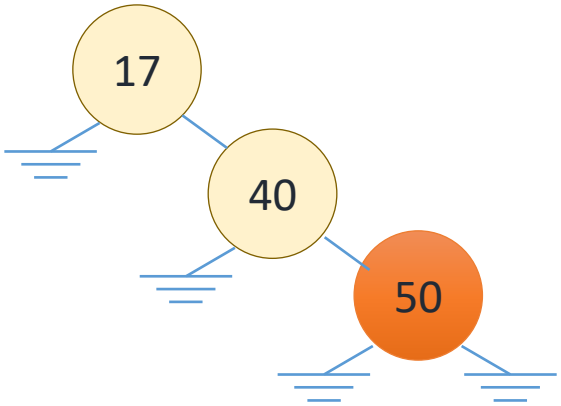


Data Structure - AVL

17, 40, 50, 49, 43, 52, 47, 45

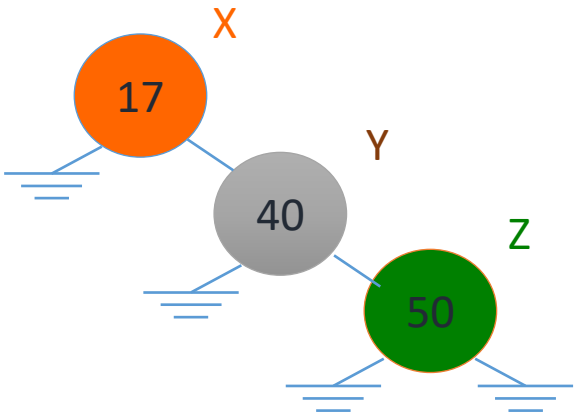


Insert 50



AVL tree becomes unbalance. Need to balance the AVL tree by a single left-rotate at node 17.

Data Structure - AVL



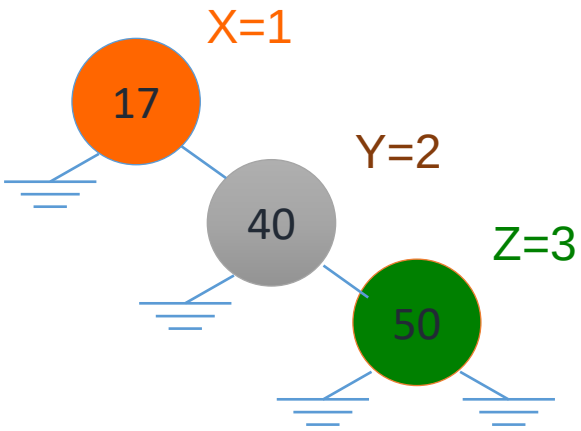
17, 40, 50, 49, 43, 52, 47, 45

Steps to identify the nodes that causes the AVL tree to become unbalance:

1. Start from the last node that is inserted to the tree, in our example, it is node 50; identify its parent and grandparent.
2. Test if the sub-tree rooted at the grandparent is balance.
   - If the sub-tree is balance, move up further, one level at a time, and check if the tree rooted at the node is balance until an unbalance sub-tree is identified.
3. Once identified the root of the sub-tree that is unbalance, label the node X, its child Y, and its grandchild Z (from the path that's used to move up in step 2.

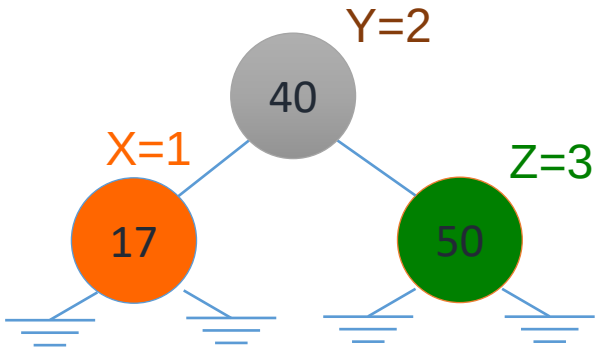
Data Structure - AVL

17, 40, 50, 49, 43, 52, 47, 45



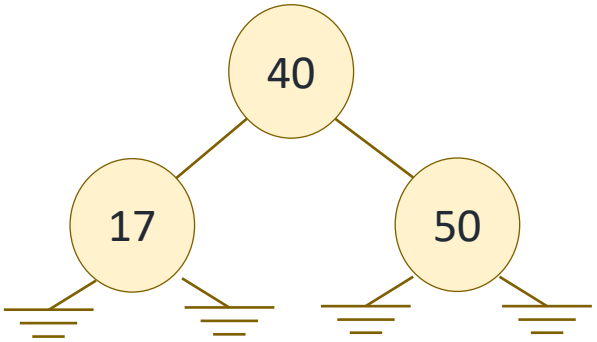
4. Traverse the three nodes, X, Y, and Z in in-order and assign the nodes with a number 1, 2, and 3 where 1, 2, and 3 is the sequence number where the nodes is visited in in-order.
5. Once identified, perform the required rotation(s) such that the node with value 2 will be the root of a sub-tree, the node with value 1 will be the root of the left-sub-tree, and the node with value 3 will be the root of the right-sub-tree.

Perform a single left-rotate at node 17, and we have →

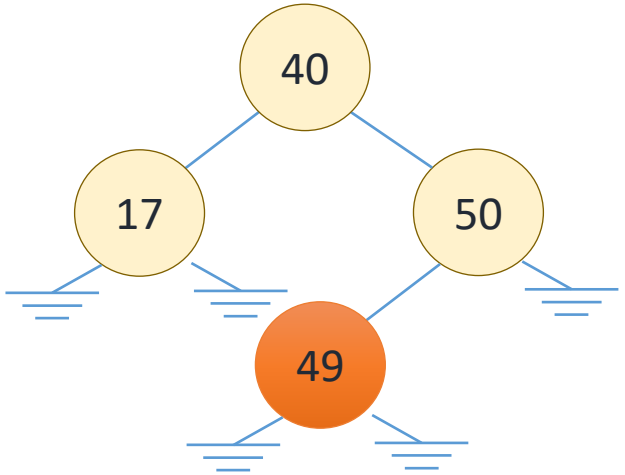


Data Structure - AVL

17, 40, 50, 49, 43, 52, 47, 45

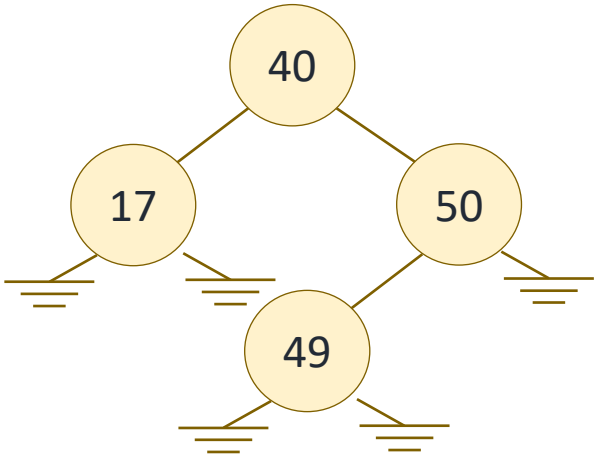


Insert 49

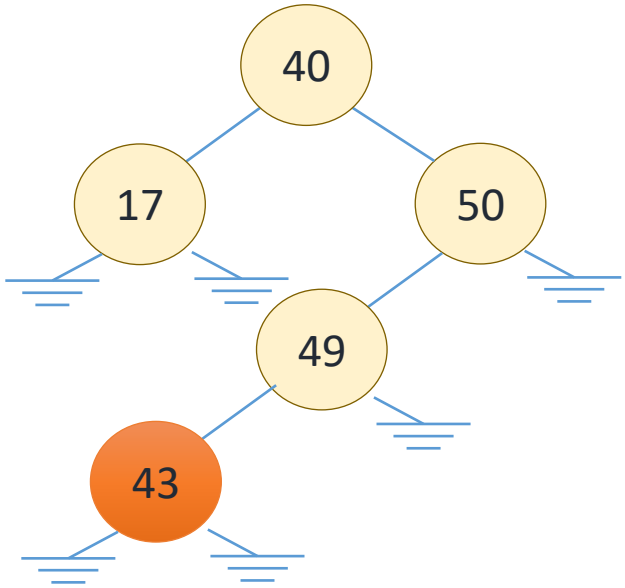


Data Structure - AVL

17, 40, 50, 49, 43, 52, 47, 45



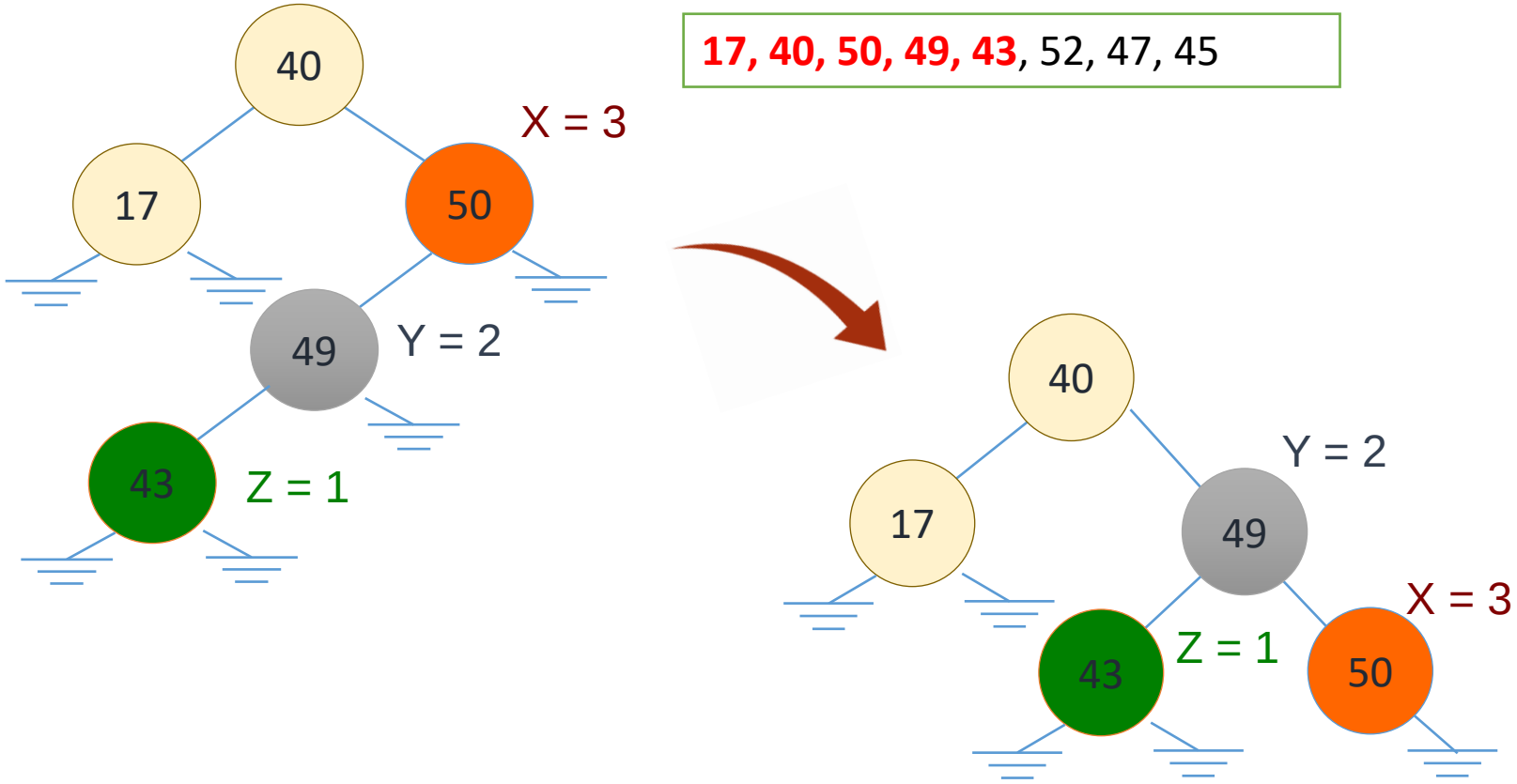
Insert 43



AVL tree becomes unbalance. Need to balance the AVL tree by a single right-rotate at node 50.

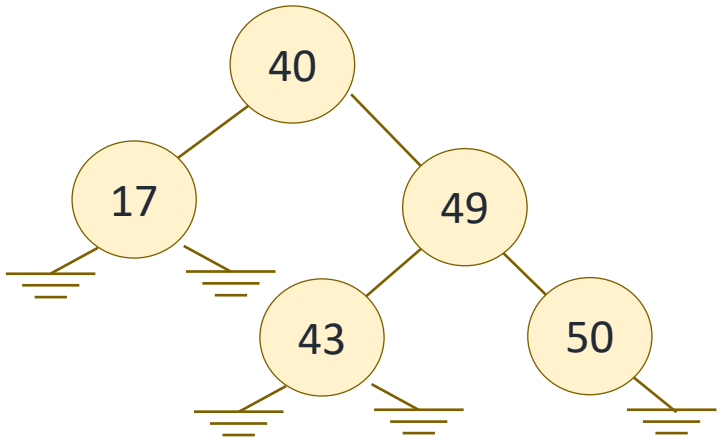
Which are the X, Y, and Z nodes?

Data Structure - AVL

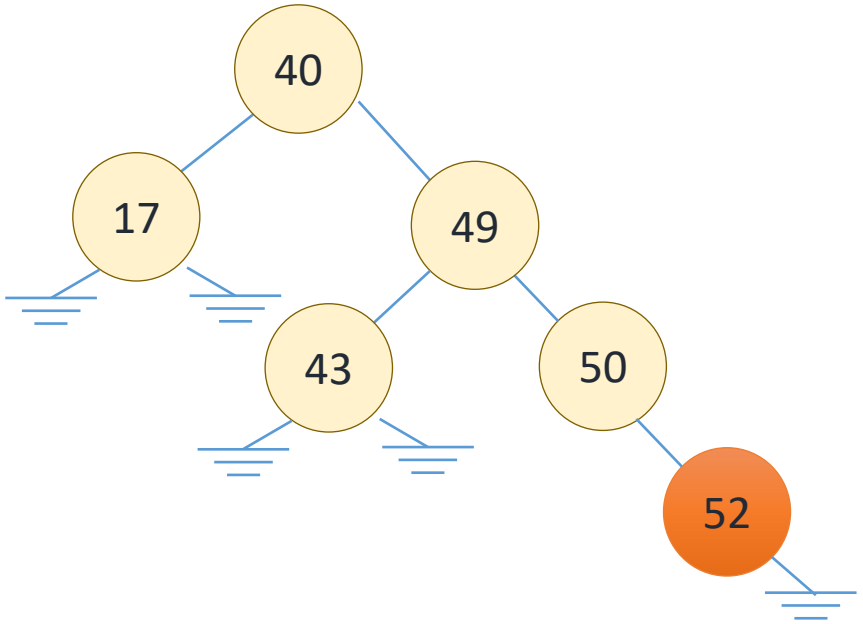


Data Structure - AVL

17, 40, 50, 49, 43, 52, 47, 45



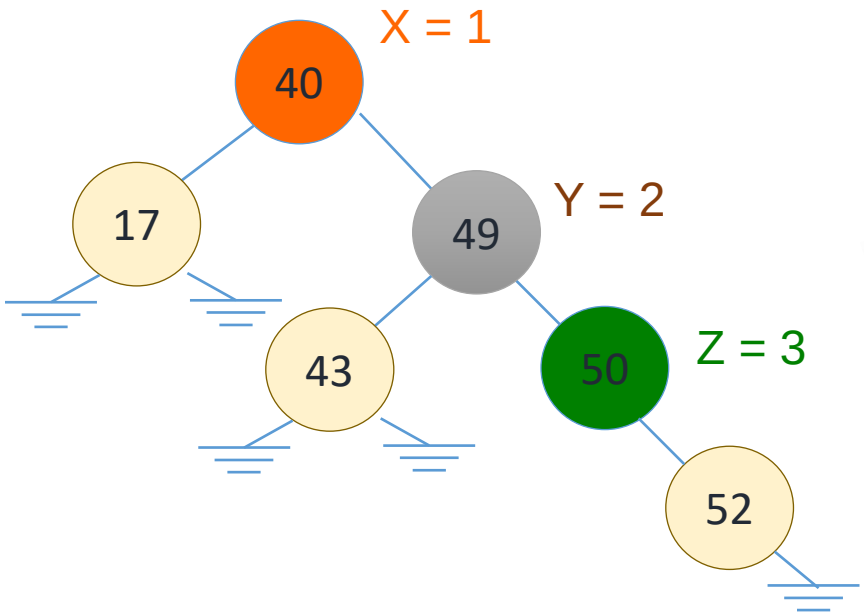
Insert 52



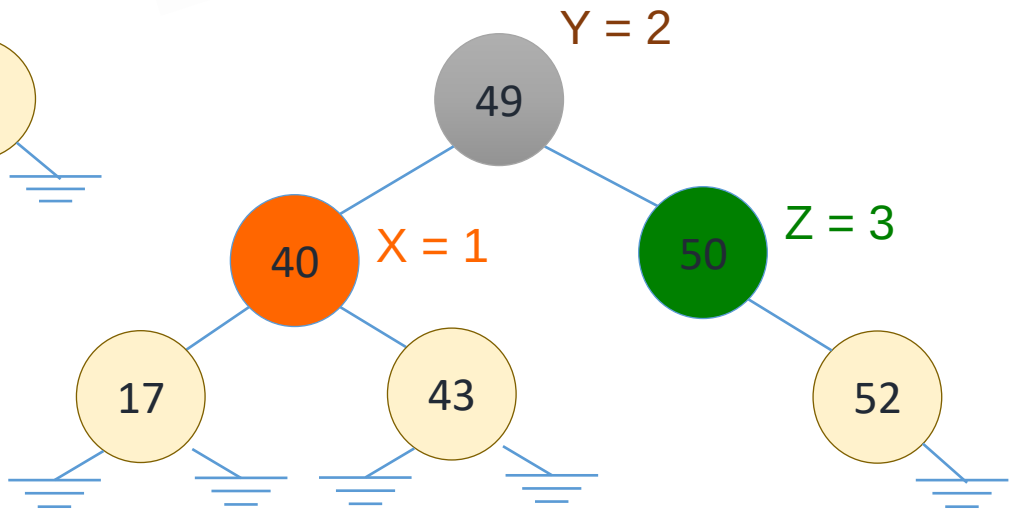
AVL tree becomes unbalance. Need to balance the AVL tree by a single complex right-rotate at node 40.

Which are the X, Y, and Z nodes?

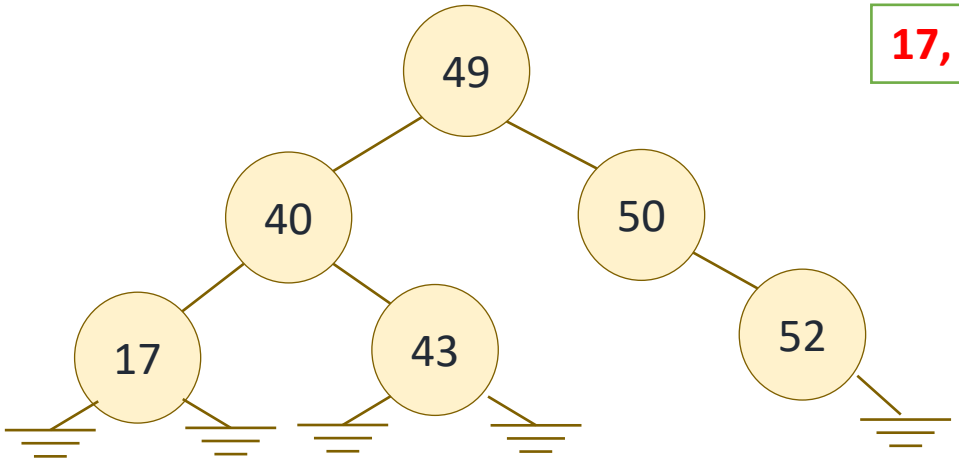
Data Structure - AVL



17, 40, 50, 49, 43, 52, 47, 45

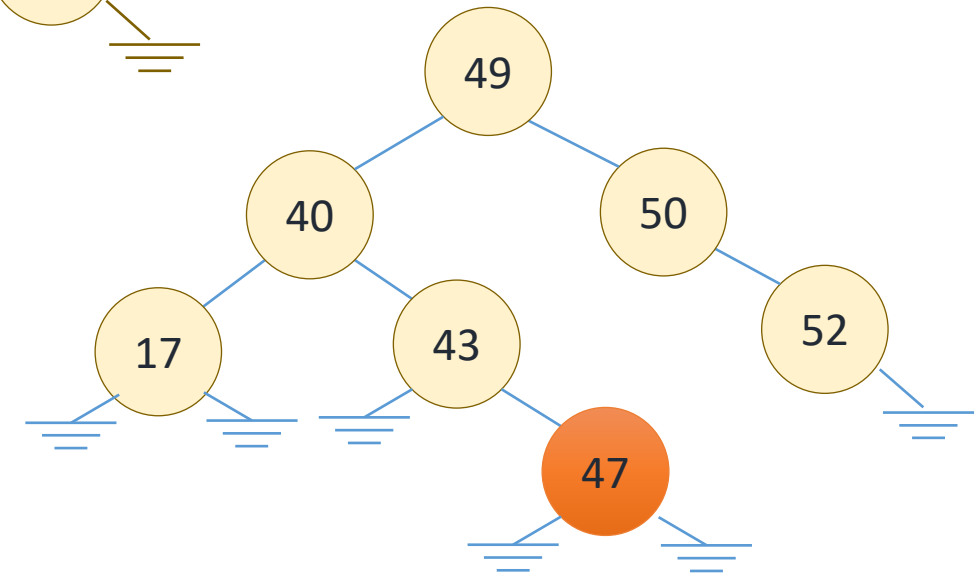


Data Structure - AVL

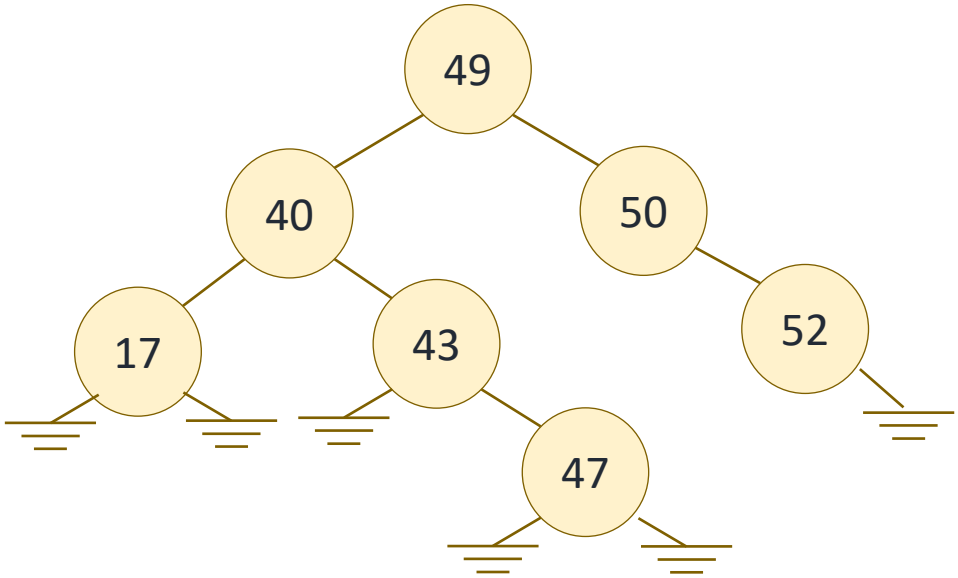


17, 40, 50, 49, 43, 52, 47, 45

Insert 47

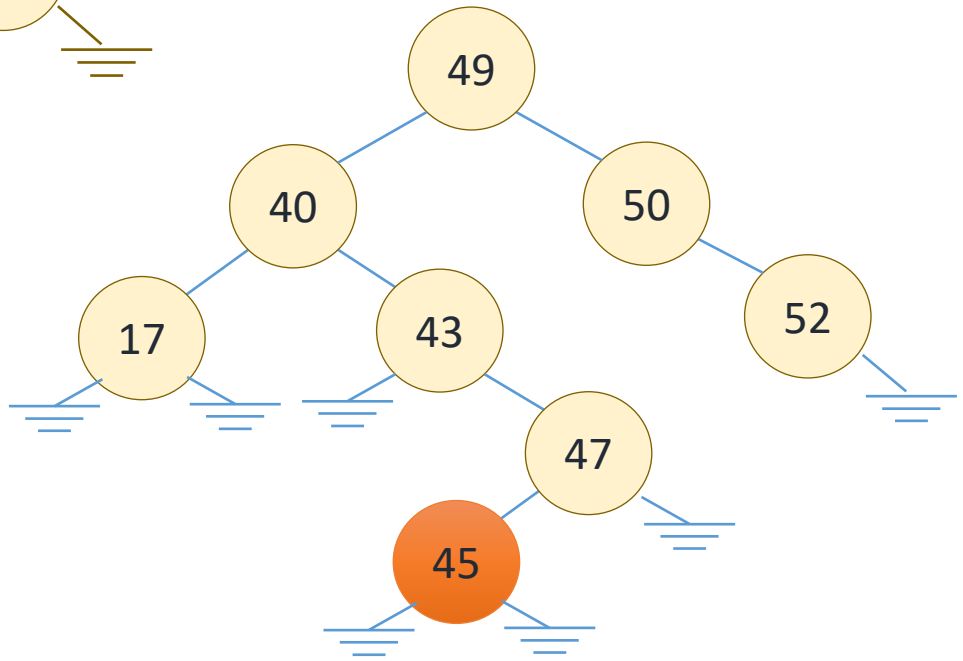


Data Structure - AVL



17, 40, 50, 49, 43, 52, 47, 45

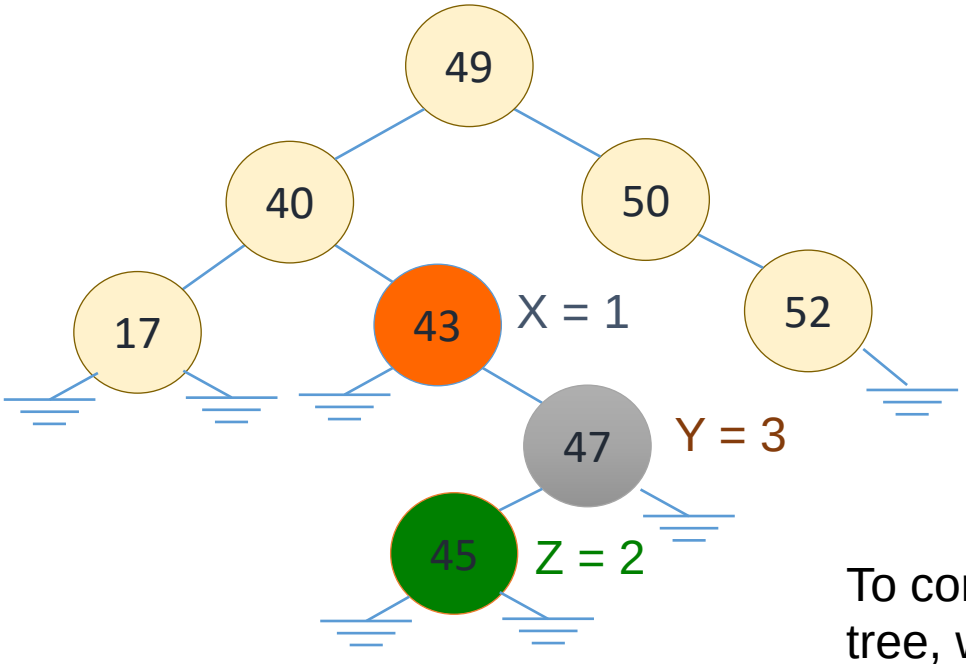
Insert 47



The AVL tree becomes unbalance after the insertion of node 45.

Which are the X, Y, and Z nodes? And what are the in-order sequence of X, Y and Z?

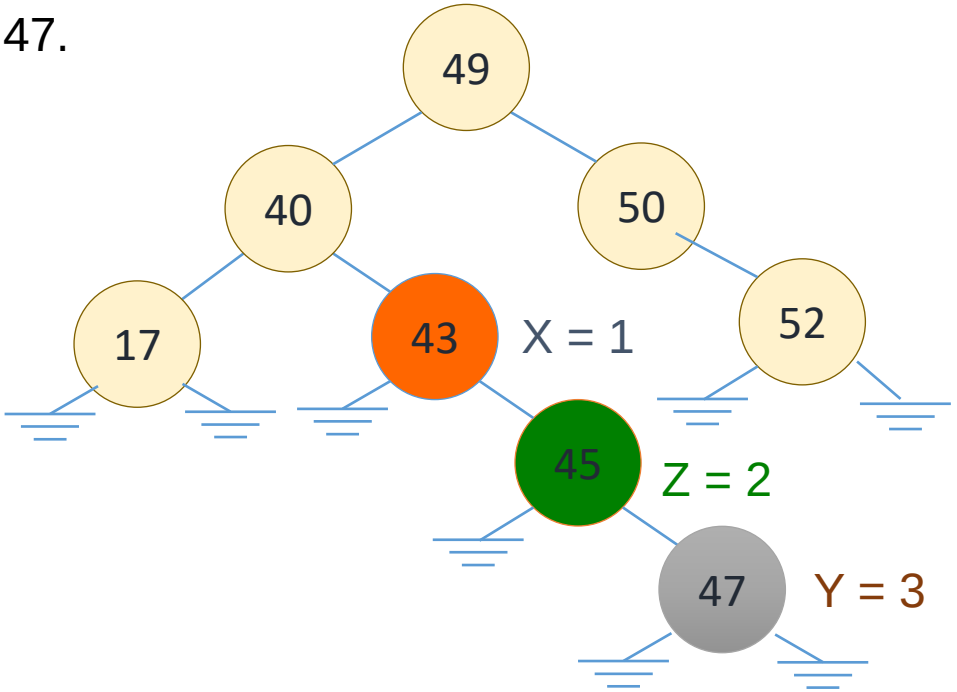
Data Structure - AVL



To correct the height of the AVL tree, we need to perform a double-right-left rotation at node 47 (rotate right) and node 43 (rotate left) respectively.

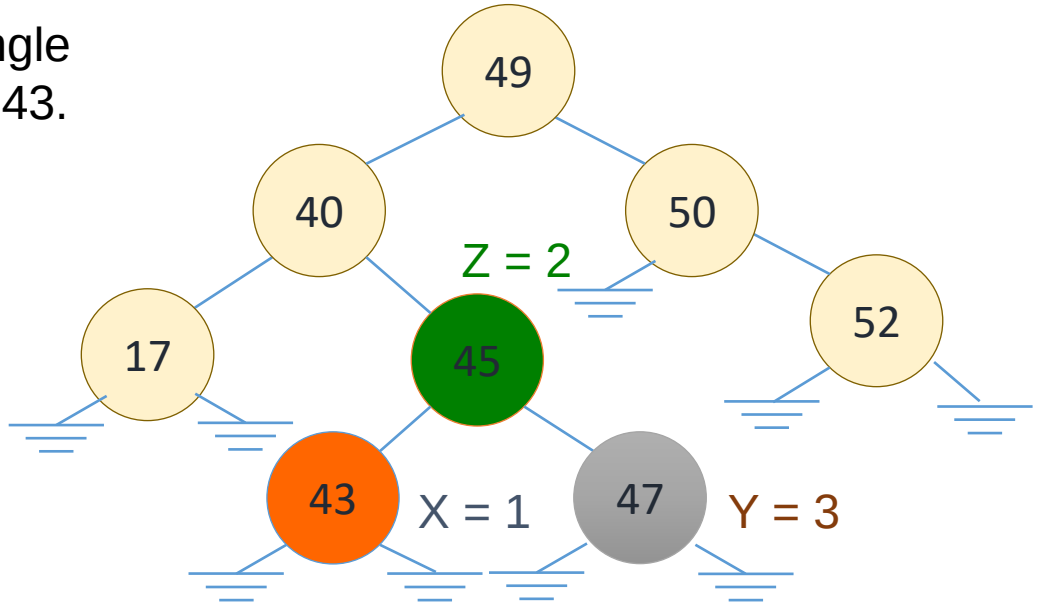
Data Structure - AVL

First perform a single right rotate at node 47.



Data Structure - AVL

Next perform a single left rotate at node 43.



Summary

- Binary tree traversal
  - Preorder (Root-Left-Right)
  - Inorder (Left-Root-Right)
  - Postorder (Left-Right-Root)
- Binary search tree operations
  - Worst case O(n)
  - Balanced tree: O(log n)
- AVL trees
  - Self balanced using rotations:
    - Single left/right rotation
    - double left/right rotation
  - Insert – 4 cases: LL, LR, RL, RR