Data Structures and Algorithms I SCS1201 - CS

Dr. Dinuni Fernando

Senior Lecturer

Lecture 9

Learning outcomes

In this topic, we will cover:

- Binary Tree – Usecase eg: Expression tree
- Tree traversals
- Concepts of:
  - Efficient binary trees
  - Binary search trees

Representing an expression using binary tree

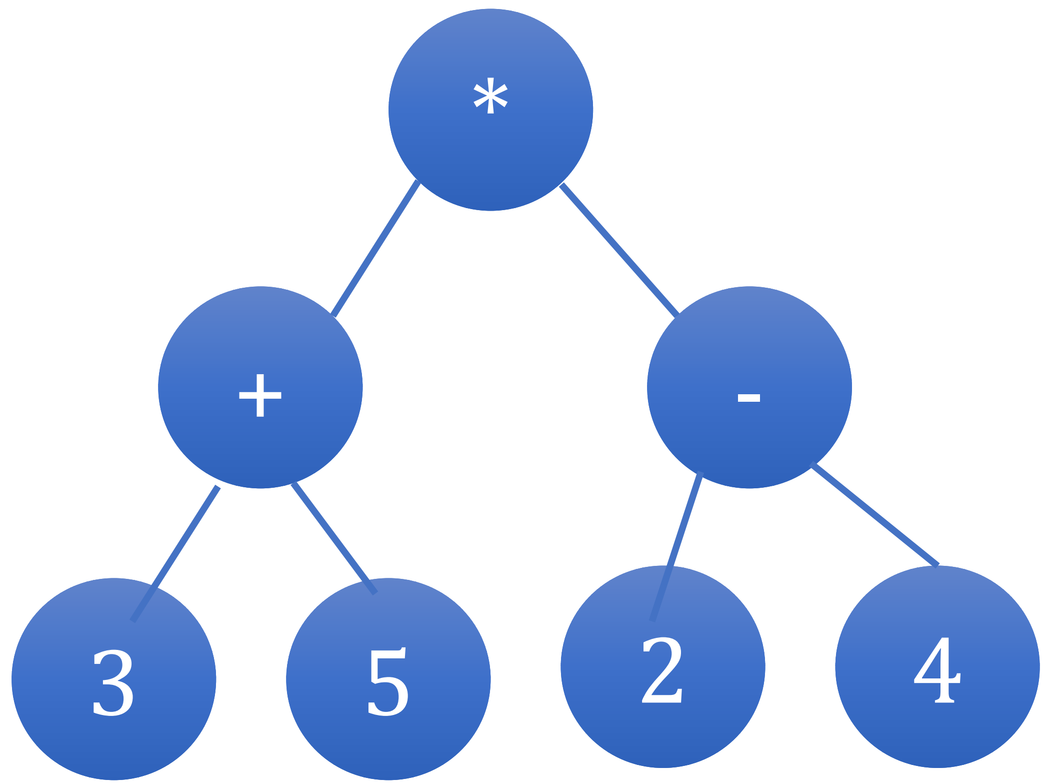
- Expression tree : is a Binary tree
- Used to represent and evaluate mathematical expressions in a hierarchical structure.
- Leaf Nodes: Contain operands (e.g., numbers, variables like x or y).
- Internal Nodes: Contain operators (e.g., +, -, *, /).
- Evaluation: The value of the expression is computed by recursively evaluating the left and right subtrees, applying the operator in the root.

Example : Expression Tree

$(3+5)*(2-4)$

Example : Expression Tree

(3+5)*(2-4)

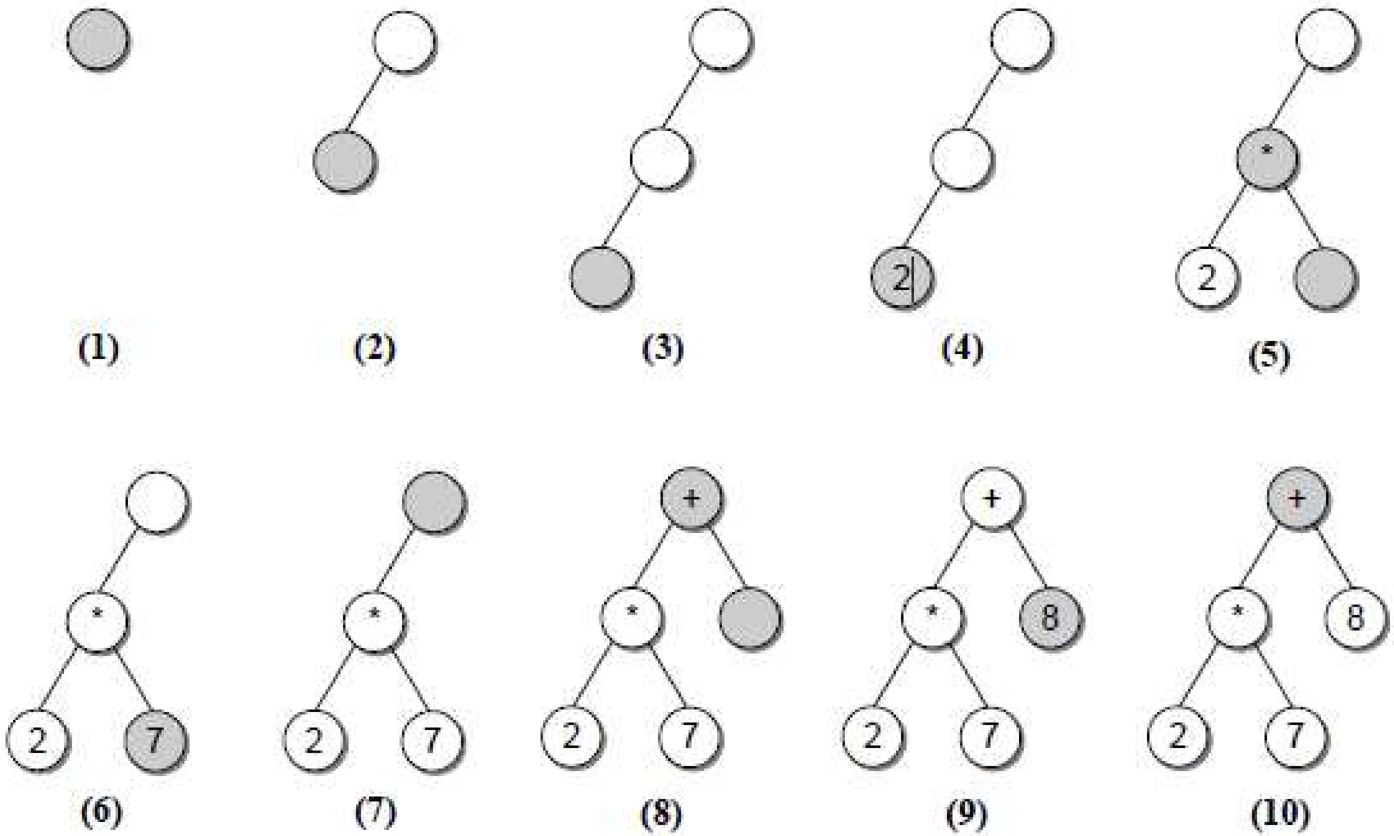


Expression tree construction

- Lets see how we can build a tree for the expression ((2*7)+8).
- Steps :
  1. Create an empty root node and mark it as the current node.
  2. Read the left parenthesis. Add a new node as the left child and descend down to the new node.
  3. Read the next left parenthesis, add a new node as the left child and descend down to the new node.
  4. Read the operand 2 : set the value of the current node to the operand and move up to the parent of the current node.
  5. Read the operator * : set the value of the current node to the operator and create a new node linked as the right child. Then descend down to the new node.
  6. Read the operand 7 : set the value of the current node to the operand and move up to the parent of the current node.

7. Read the right parenthesis : move up to the parent of the current node.
8. Read the operator + : set the value of the current node to the operator and create a new node linked as the right child , descend down to the new node.
9. Read the operand 8: set the value of the current node to the operand and move up to the parent of the current node.
10. Read the right parenthesis: move up to the parent of the current node. Since this is the last token, we are finished, and the expression tree is complete.

Steps for building an expression tree for ((2 * 7) + 8).



Tree Traversals

- A means of visiting all the objects in a tree data structure
- We will look at
  - Breadth-first traversals
  - Depth-first traversals
- Applications
- General guidelines

Background

All the objects stored in an array or linked list can be accessed sequentially.

Question: how can we iterate through all the objects in a tree in a predictable and efficient manner

- Requirements: $\Theta(n)$ run time and $o(n)$ memory

Types of Traversals

- The breadth-first traversal visits all nodes at depth k before proceeding onto depth $k + 1$ (aka Level order traversal)
  - Easy to implement using a queue

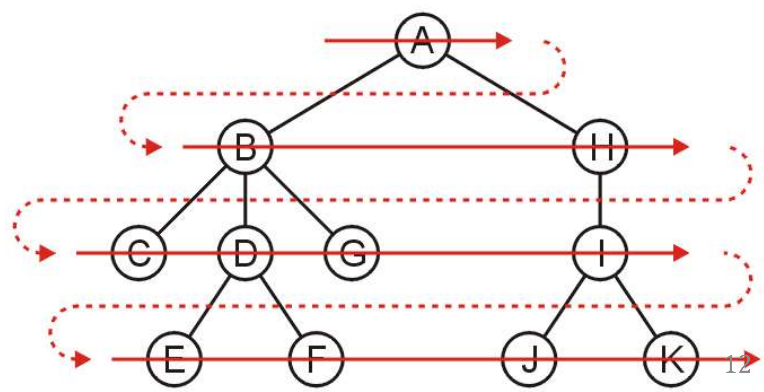
- Another approach is to visit always go as deep as possible before visiting other siblings: *depth-first traversals*
  - *In-order traversal*
  - *Pre-order traversal*
  - *Post-order traversal*

Breadth-First Traversal

Breadth-first traversals visit all nodes at a given depth

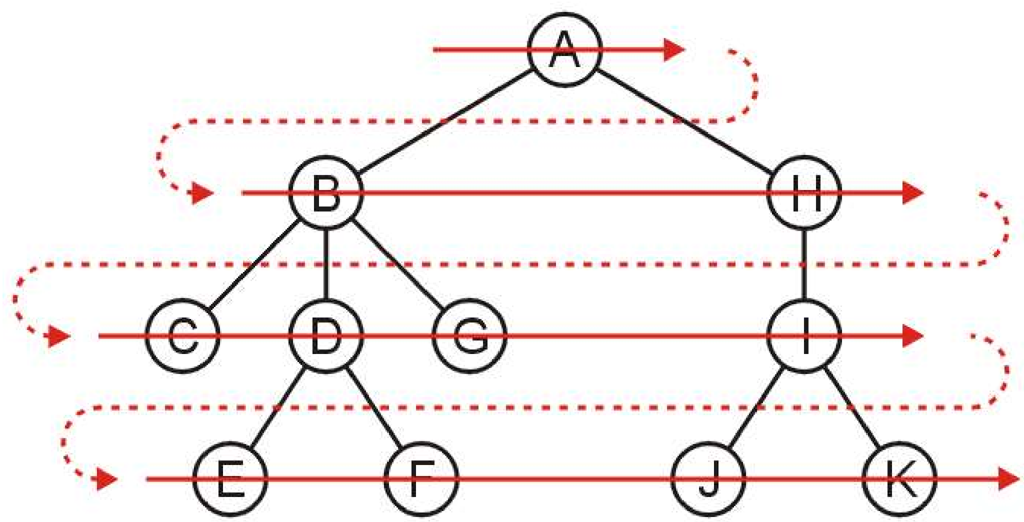
Idea : expand a frontier one step at a time.

- Can be implemented using a queue (FIFO)
- Run time is Q(n)
- Memory is potentially expensive: maximum nodes at a given depth
- Order: A B H C D G I E F J K



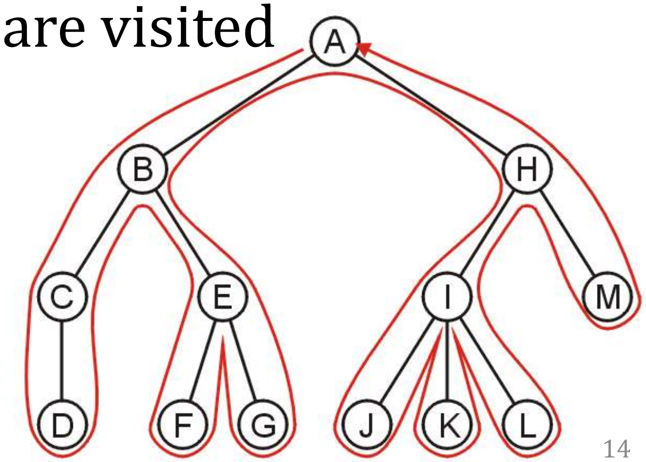
Breadth-First Traversal

- Create a queue and push the root node onto the queue
- While the queue is not empty:
  - Push all of its children of the front node onto the queue
  - Pop the front node

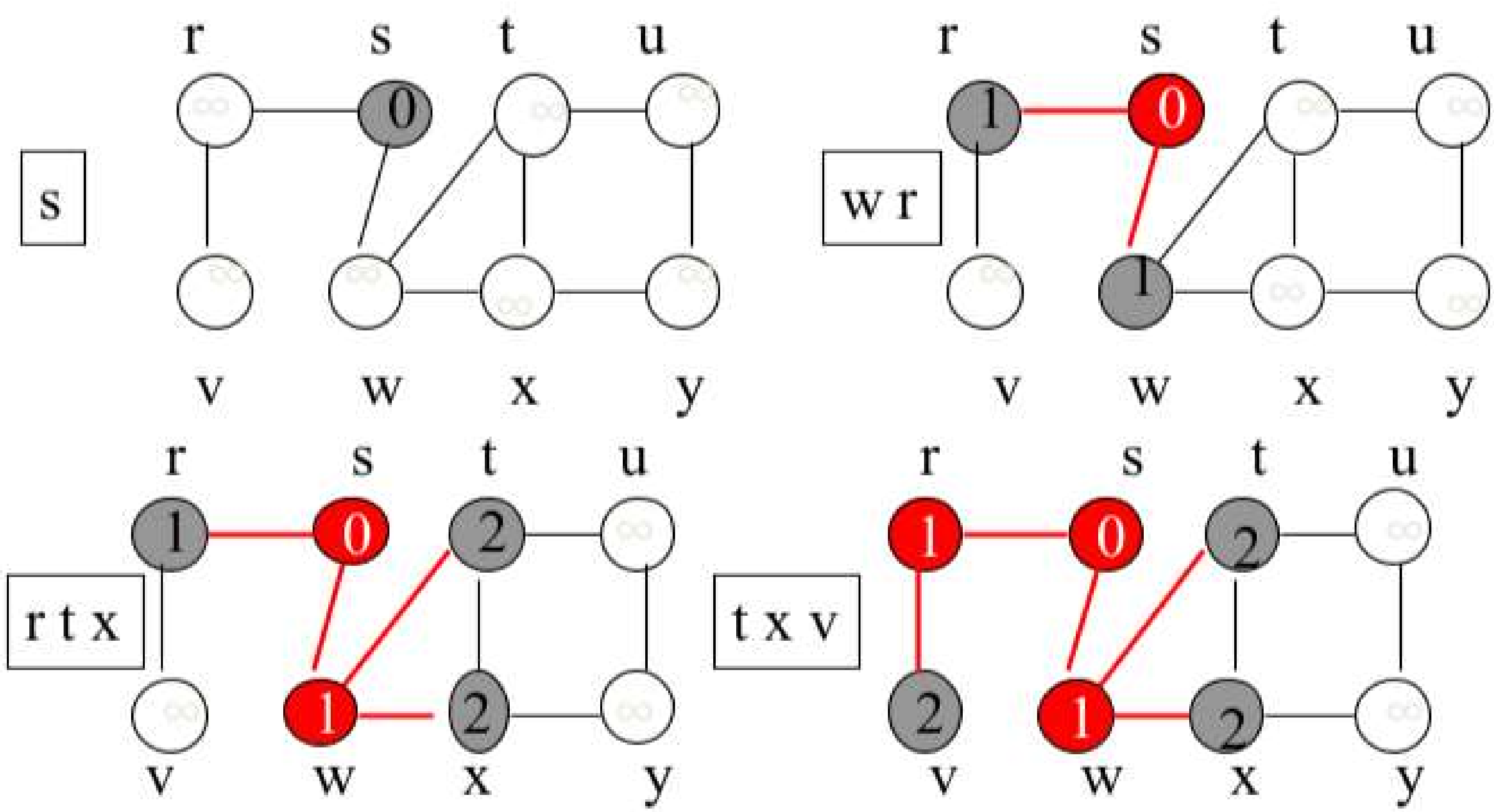


Backtracking Algorithm

- At any node, we proceed to the first child that has not yet been visited
- Or, if we have visited all the children (of which a leaf node is a special case), we backtrack to the parent and repeat this decision making process.
- We end once all the children of the root are visited

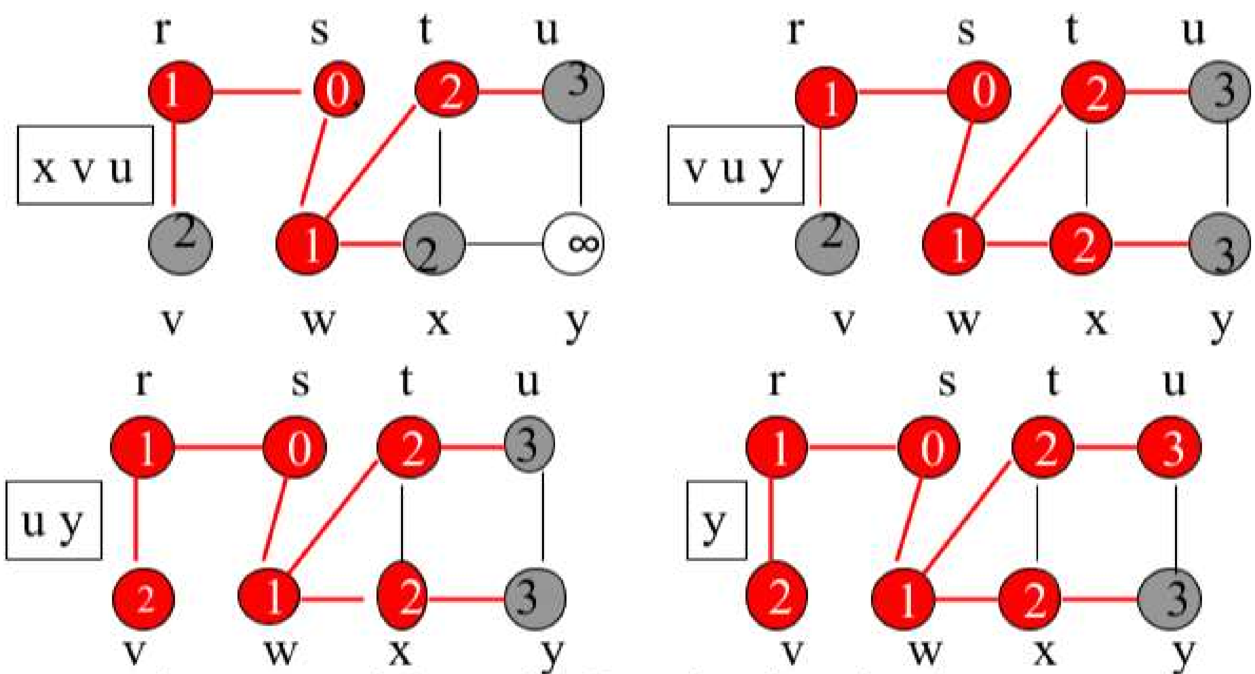


BFS – Example



r s t u
s
0
v w x y
r s t u
1 0
w r
1
v w x y
r s t u
1 0 2
r t x
1 2
v w x y
r s t u
1 0 2
t x v
2 1 2
v w x y

BFS – Example



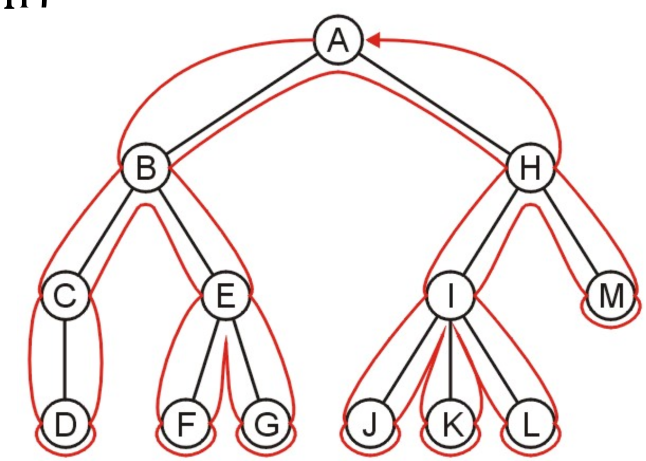
now y is removed from the Q and colored red

Depth-first Traversal

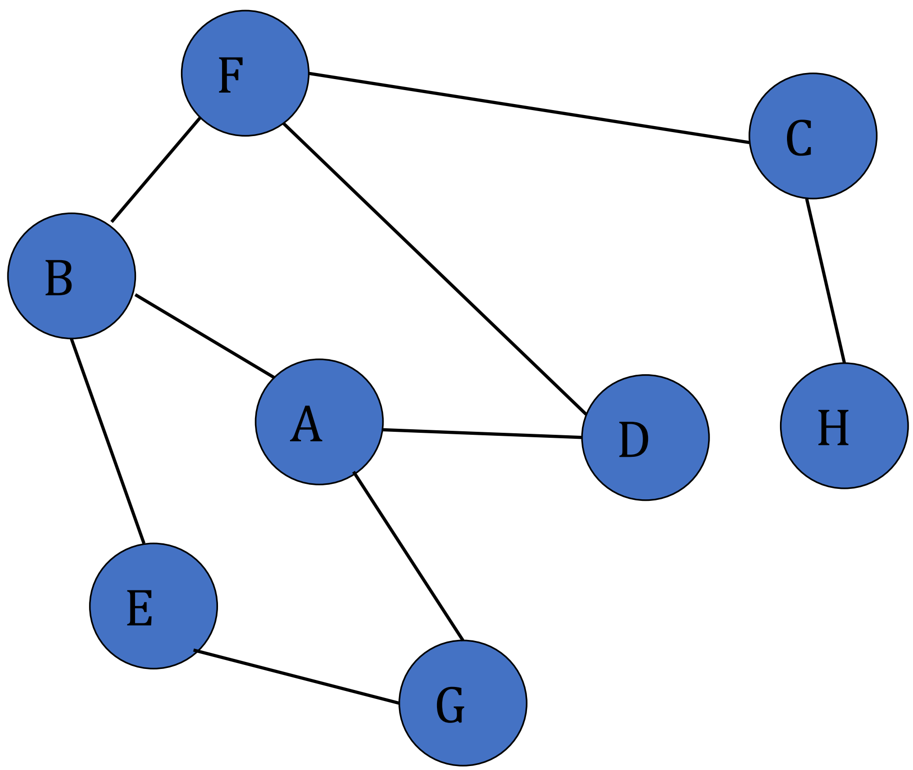
Idea : Explore every node and edge of the graph. We go deeper whenever is possible.

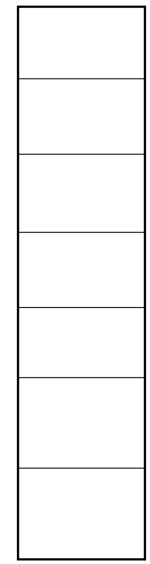
We note that each node could be visited twice in such a scheme

- The first time the node is approached (before any children)
- The last time it is approached (after all children)



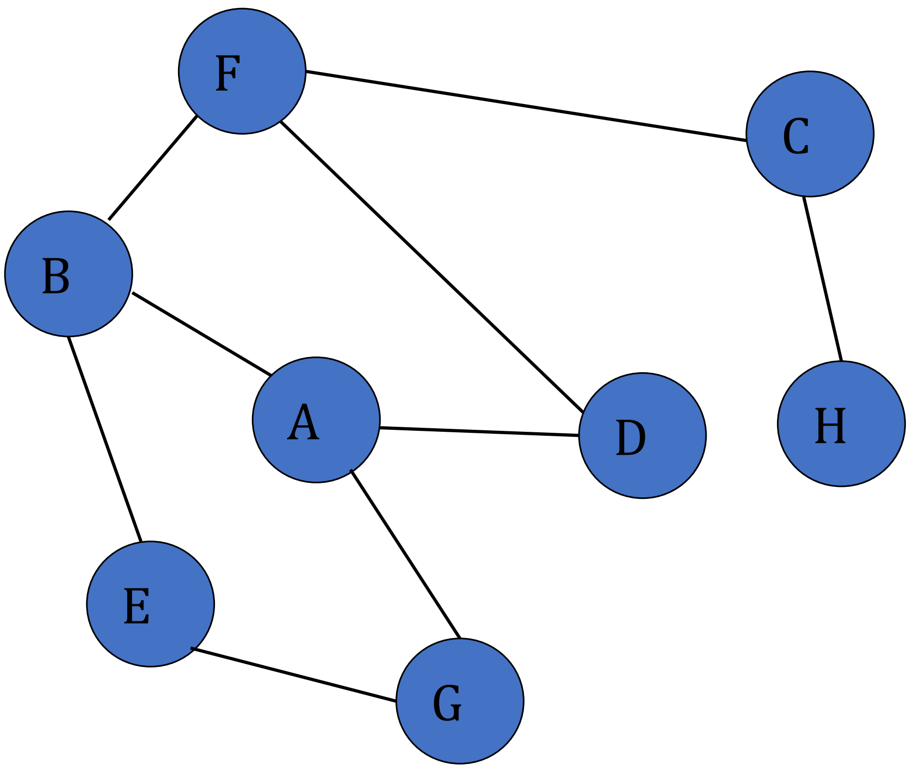
Depth-first Traversal

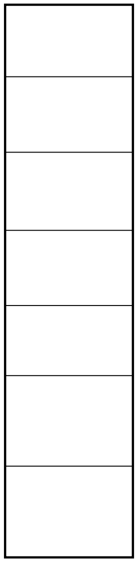




Stack

Depth-first Traversal

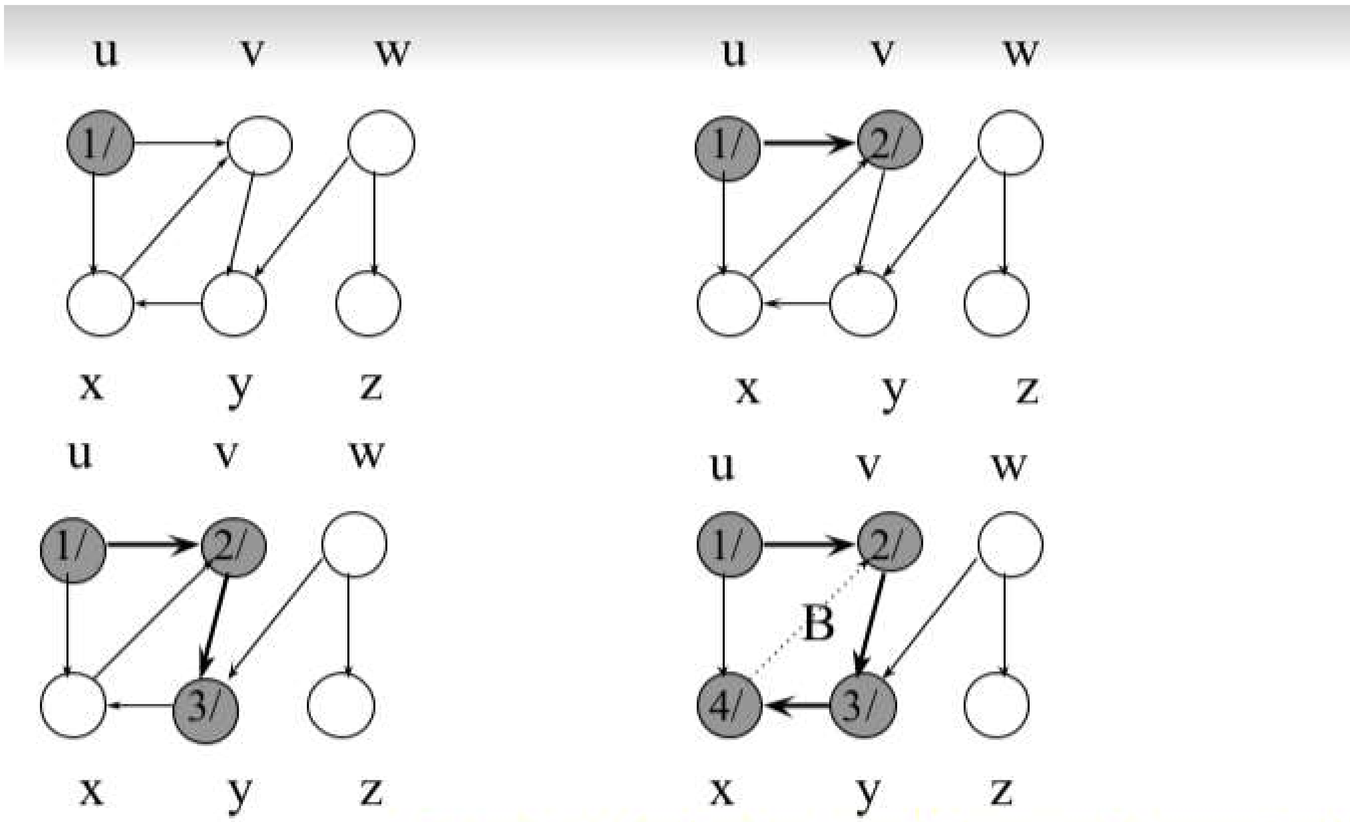




Stack

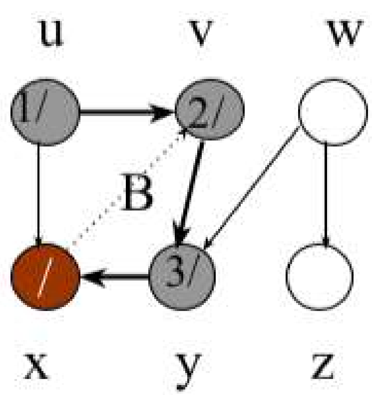
Output : ABEGDFCH

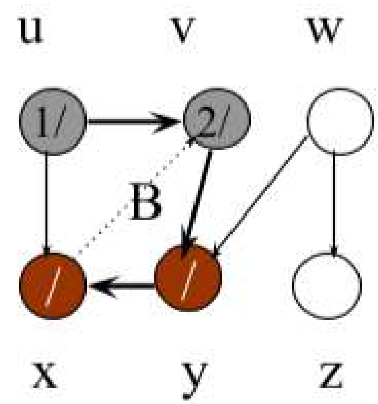
Depth-first Traversal



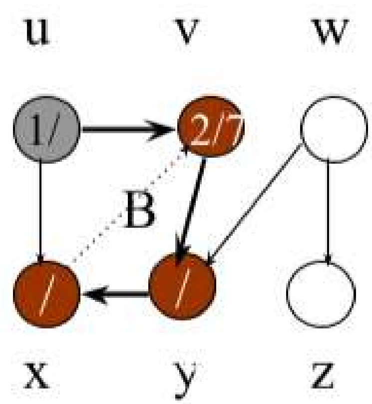
B: Back edge (edge from a node to one of its ancestors)

Depth-first Traversal





B: back edge (edge from a node to one of its ancestors)

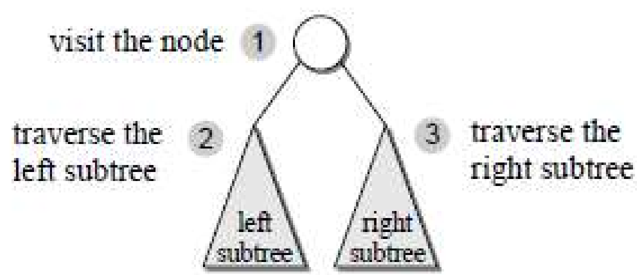


DFS based Tree Traversals Techniques

- A traversal of a tree is a systematic way of accessing or “visiting” all the nodes in the tree.
- These DFS based traversal techniques explore tree deeply as possible before backtracking.
  - Use recursive or stack for iterative implementations.
  - Operate based on order in which nodes are visited relative to their parent.
- There are three basic traversal schemes:
  - Pre-order traversal
  - In-order traversal
  - Post-order traversal

Pre-Order Traversal

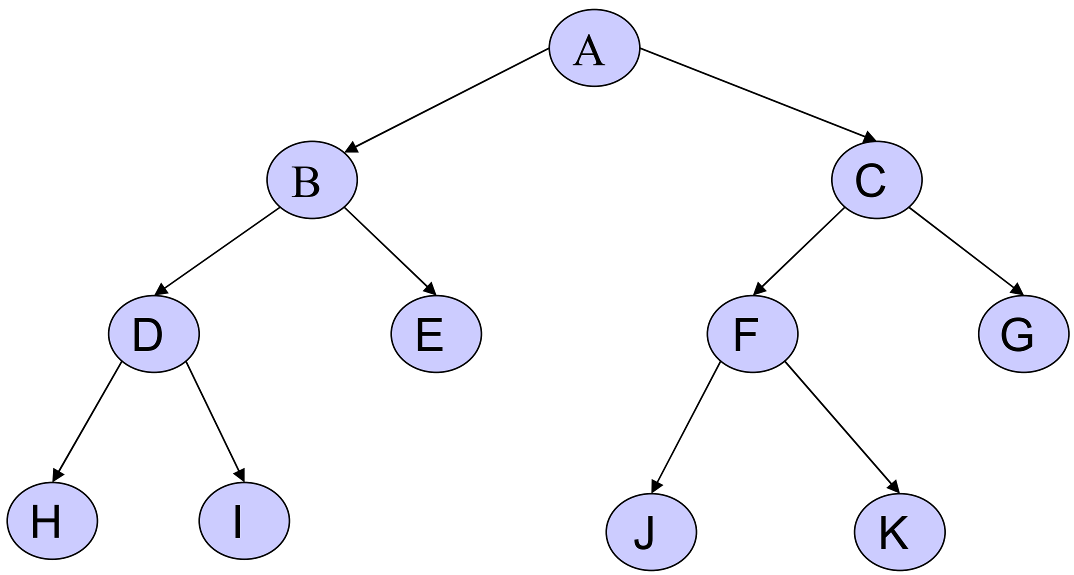
- A pre-order traversal has three steps for a nonempty tree:
  - Process the root.
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.



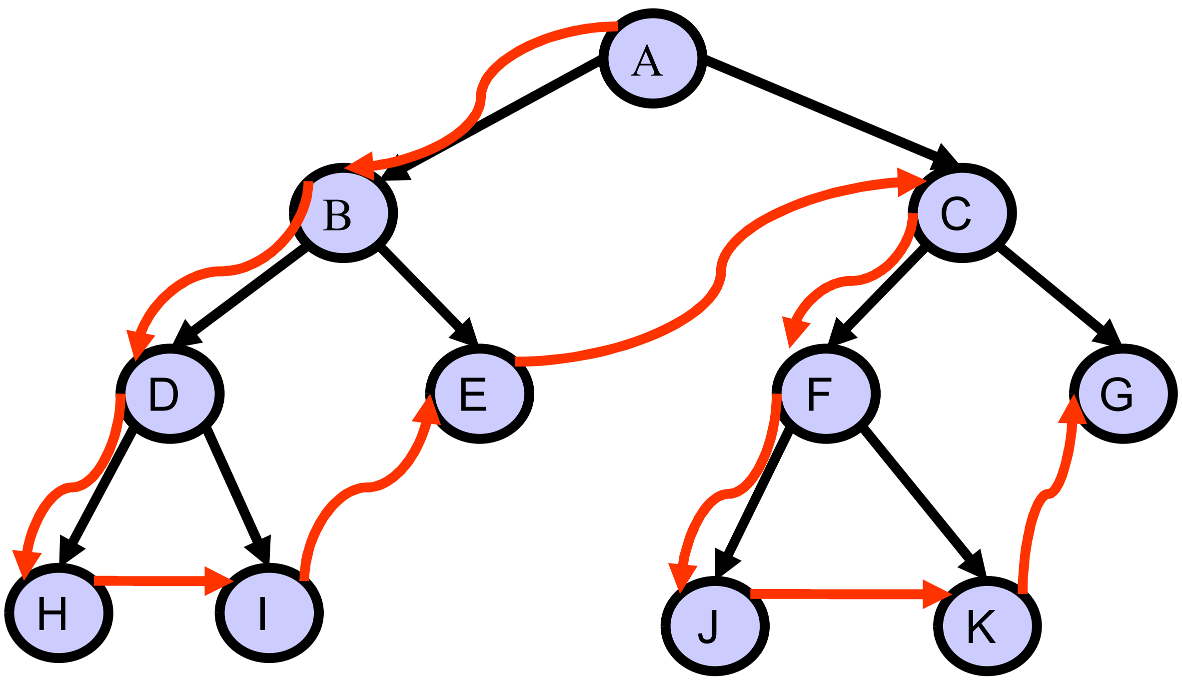
Pre-Order Traversal

- To traverse a non-empty binary tree in pre-order (also known as depth first order), we perform the following operations.
- Visit the root (or print the root)
- Traverse the left in pre-order (Recursive)
- Traverse the right tree in pre-order (Recursive)

Pre-Order Traversal



Pre-Order Traversal



Algorithm for pre-order traversal

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

In-order Traversal

- An in-order traversal has three steps for a nonempty tree:
  - Process the nodes in the left subtree with a recursive call.
  - Process the root.
  - Process the nodes in the right subtree with a recursive call.

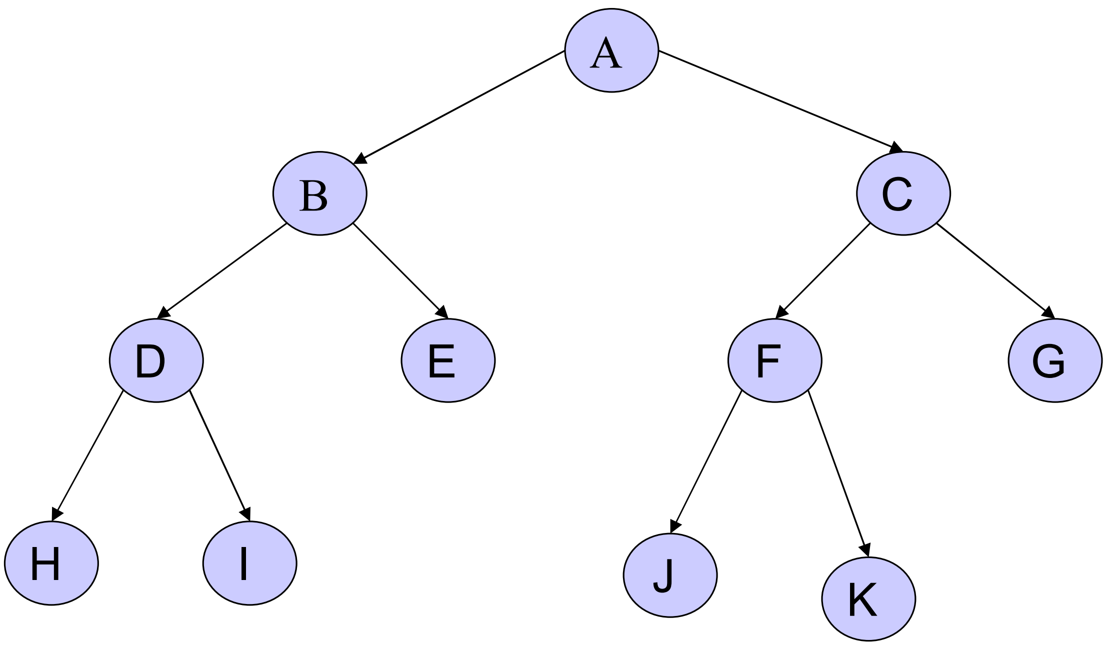
In-order traversal

The in-order listing of the nodes of T is the nodes of T_1 in in-order, followed by n,followed by the nodes T_1,T_2,....T_n, each group of nodes in in-order.

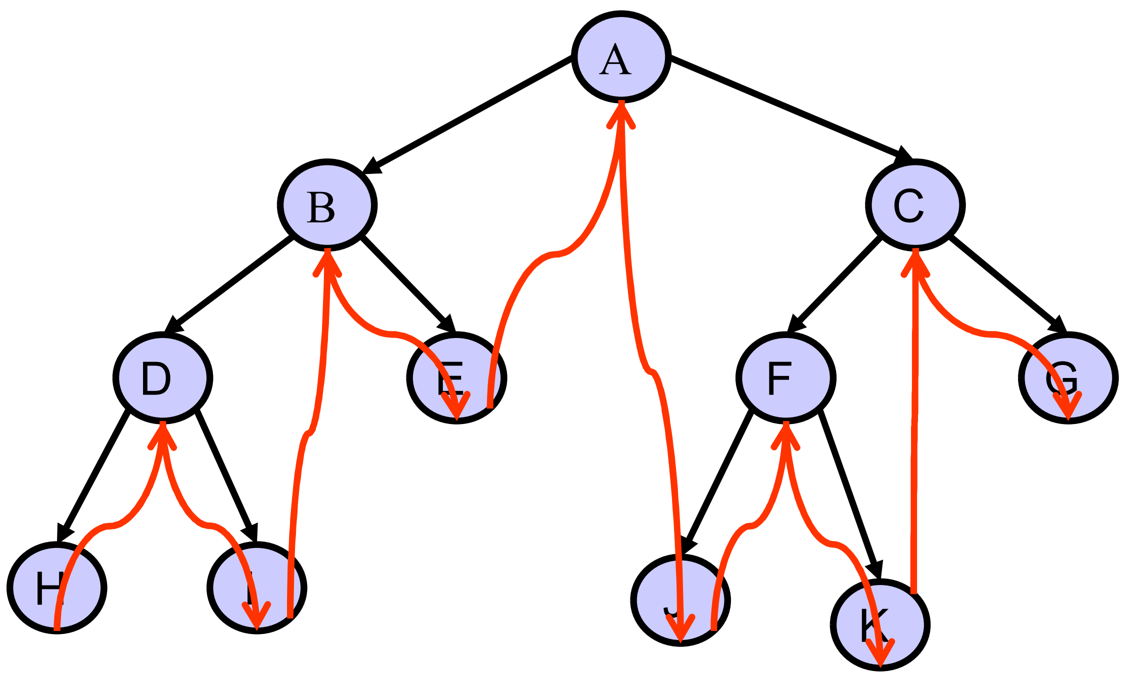
Algorithm :

- Traverse the left-subtree in in-order
- Visit the root
- Traverse the right-subtree in in-order.

In-order traversal



In-order traversal



Algorithm for in-order traversal

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

Post-order Traversal

- A post-order traversal has three steps for a nonempty tree:
  - Process the nodes in the left subtree with a recursive call.
  - Process the nodes in the right subtree with a recursive call.
  - Process the root.

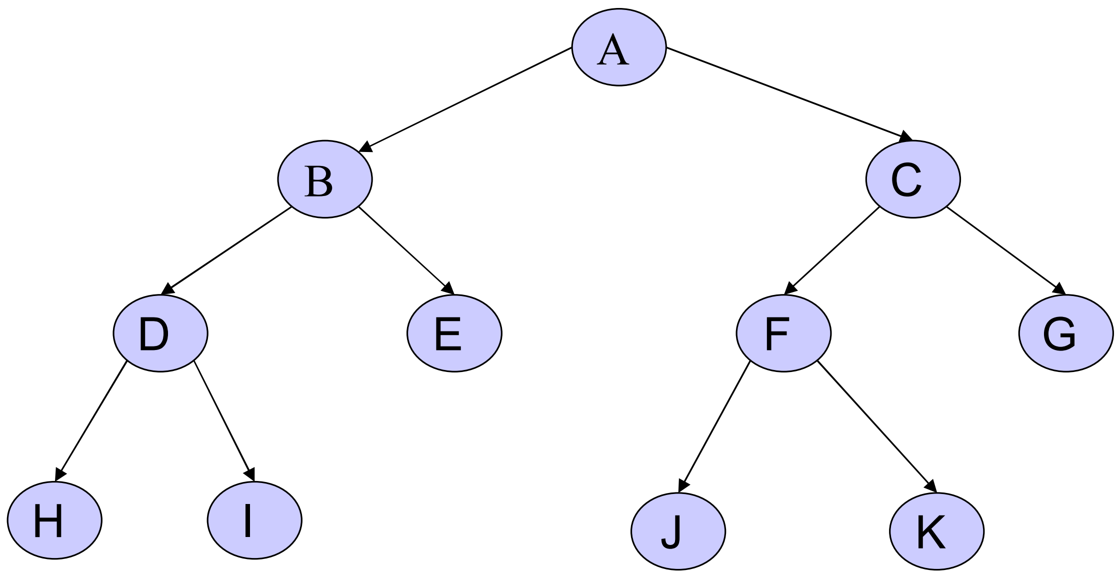
Post-order Traversal

The post-order listing of the nodes of T is the nodes of T_1 in post-order, then the nodes of T_2 in post order and so-on up to T_k, all followed by n

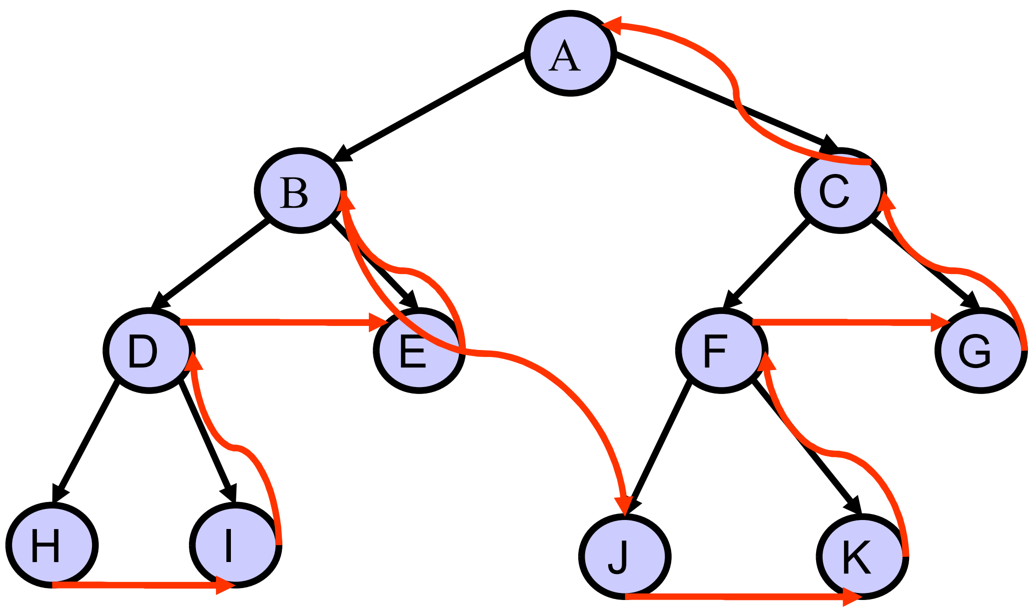
Algorithm

- Travers the left sub-tree in post-order
- Traverse the right sub-tree in post-order
- Visit the root

Post-order Traversal



Post-order Traversal



Algorithm for post-order traversal

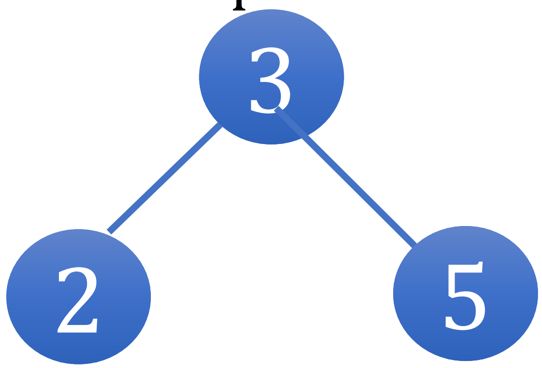
```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             POSTORDER(TREE -> LEFT)
Step 3:             POSTORDER(TREE -> RIGHT)
Step 4:             Write TREE -> DATA
        [END OF LOOP]
Step 5: END
```

Efficient Binary Trees

- This is an extension of binary trees.
- The efficient binary trees are binary search trees, AVL trees, threaded binary trees, red-black trees, and splay trees.
- Under SCS1308 we will cover Binary search trees ,AVL trees and red black trees.

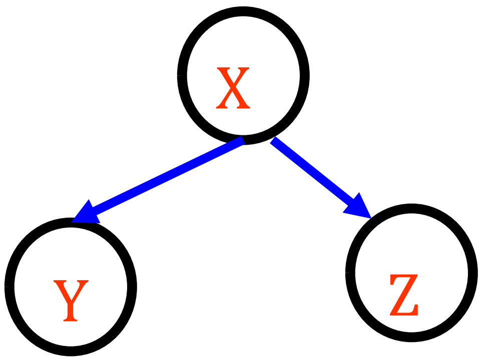
Binary Search trees

- Also known as ordered binary tree
  - variant of binary trees where nodes are arranged in an order.
- In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Similarly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree.
- (Note that a binary search tree may or may not contain duplicate values, depending on its implementation.)

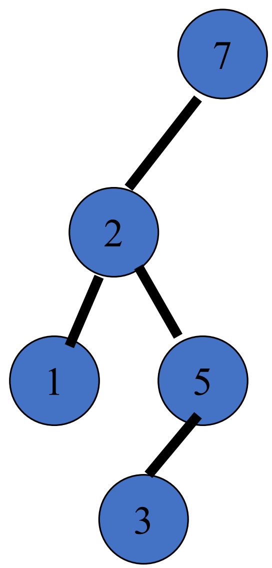


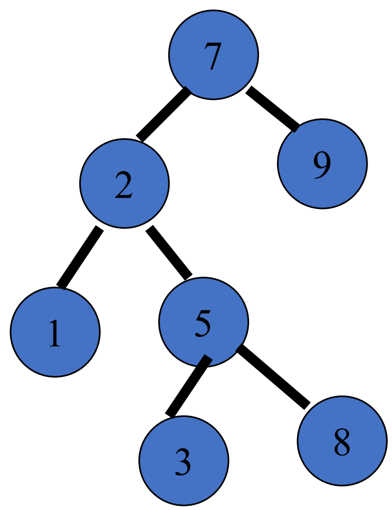
Binary Search trees

- Key property
  - Value at node
    - Smaller values in left subtree
    - Larger values in right subtree
  - Example
    - X > Y
    - X < Z

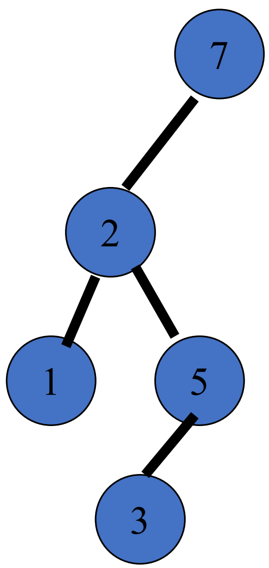


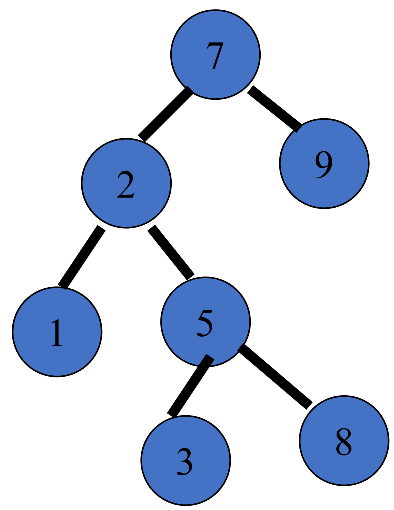
Binary Search trees (Examples)



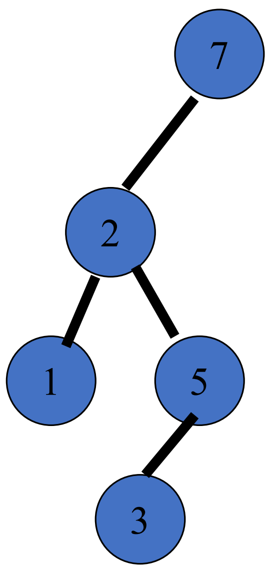


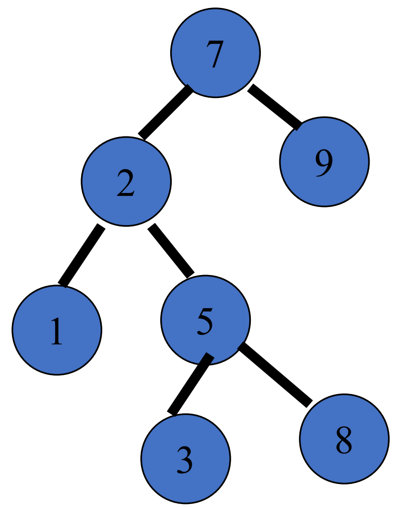
Binary Search trees (Examples)





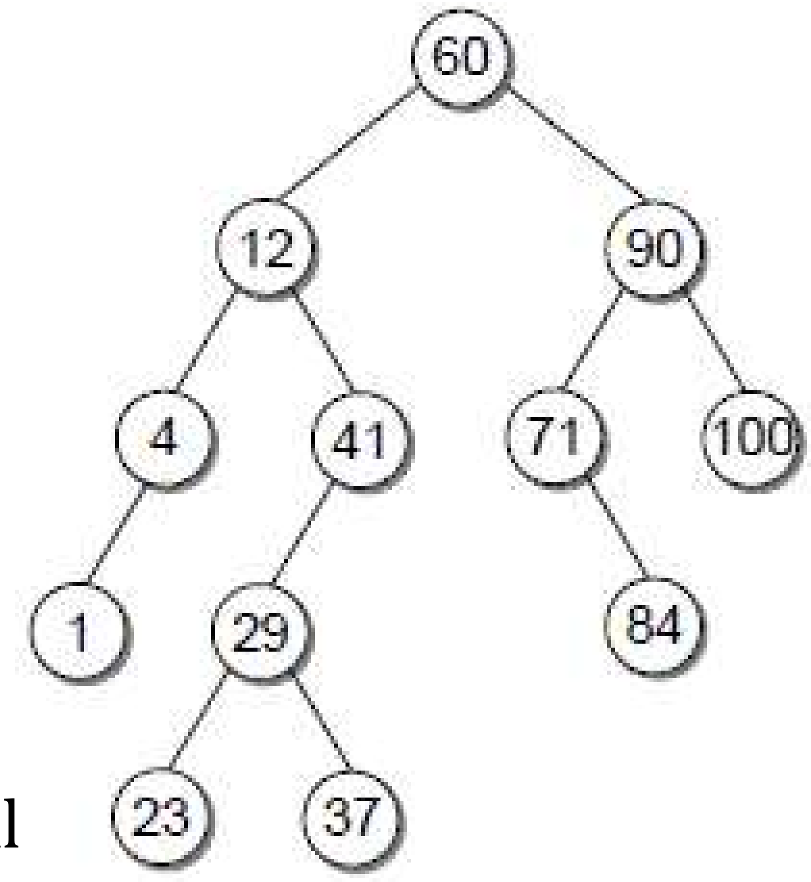
Binary Search trees (Examples)





Two binary trees(only the left tree
is a search tree)

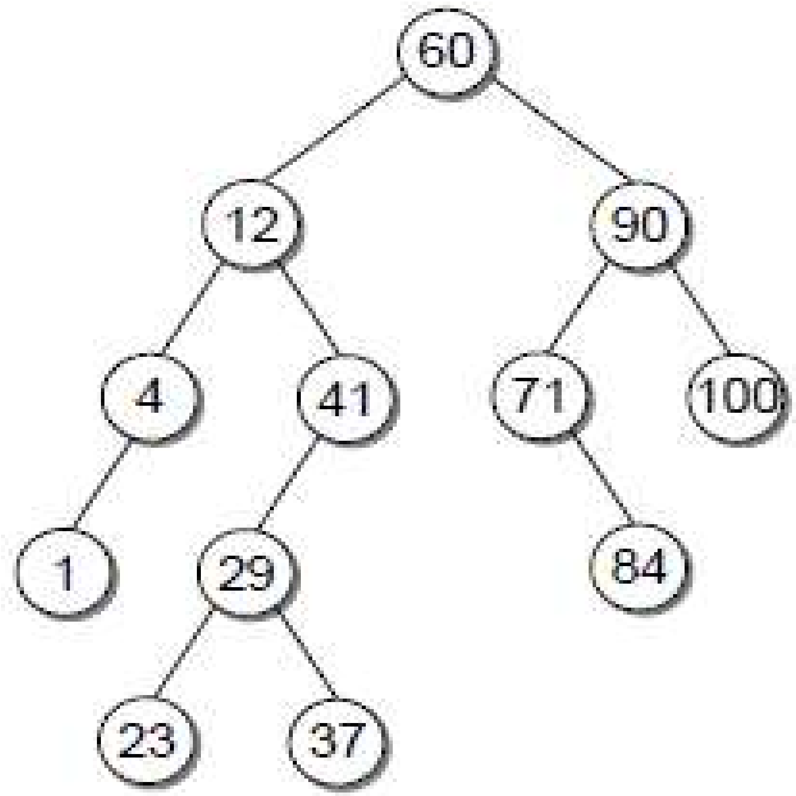
Example : Binary Search trees



What is the in-order traversal of the above tree ?

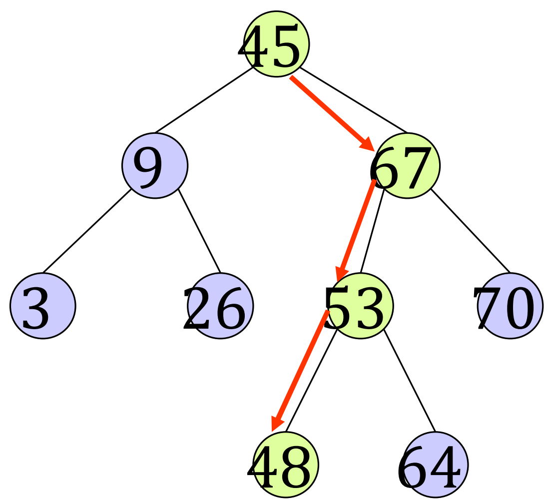
What is the in-order traversal of the above tree ?

The order would be 1 4 12 23 29 37 41 60 71 84 90 100.



Binary Search trees

- Advantage of using binary search trees
  —support fast search.



Lets Find an element (48) in the binary search tree

Binary Search trees : search

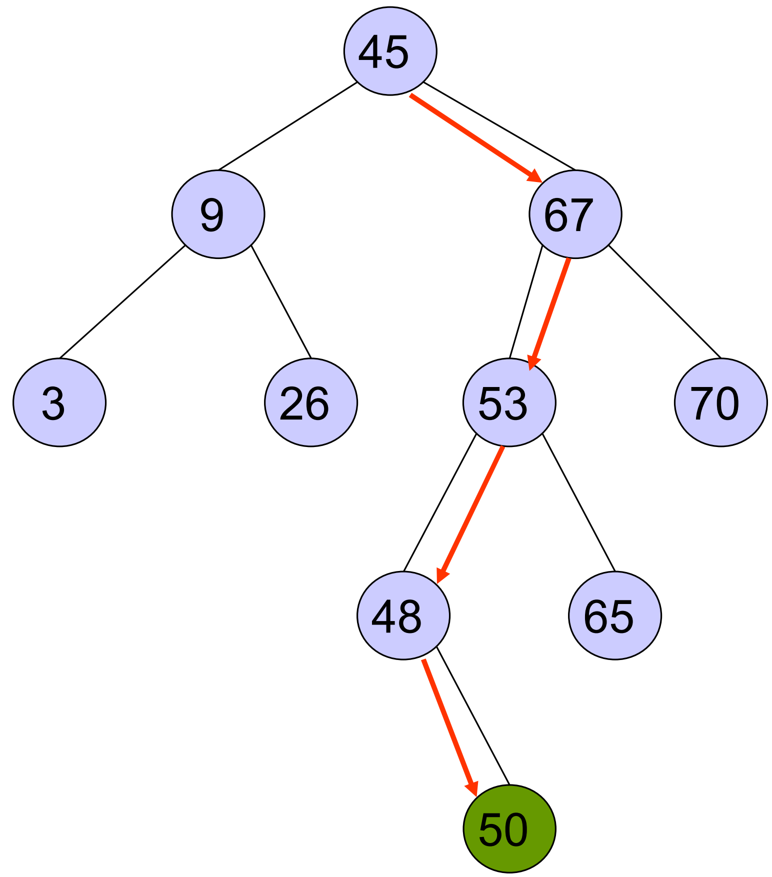
- Algorithm to search for a given value in BST

```
searchElement (TREE, VAL)

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

Binary Search trees

- Adding a new element to a binary search tree.
- Example: add a new element 50 to the binary search tree.



Binary Search trees : insert

- Algorithm to insert a given value in a binary search tree

```
Insert (TREE, VAL)

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

Binary Search trees : Applications

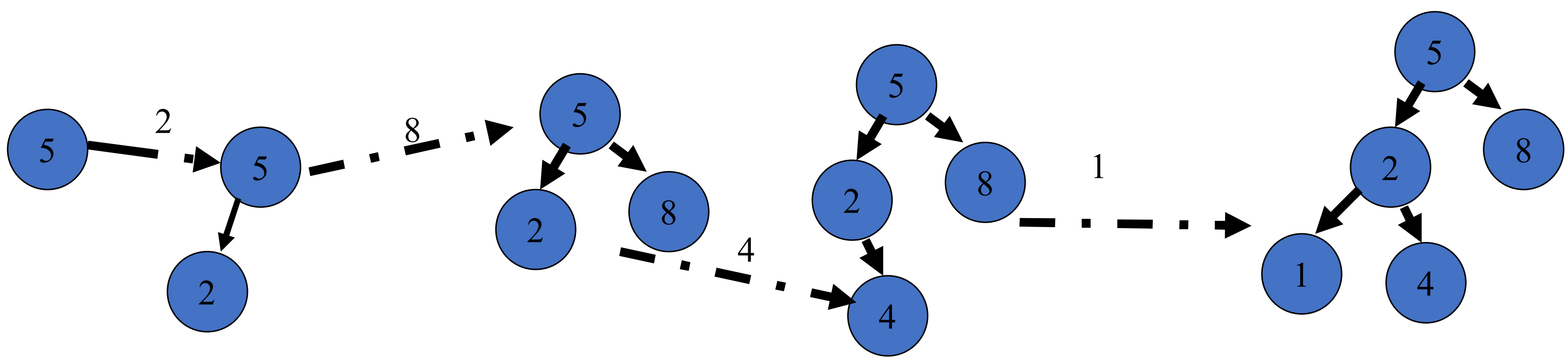
To sort data

- We have list of integers, such as 5,2,8,4, and 1 that we wish to sort in to ascending order.
- This algorithm consists of two stages.
  - Insertion
  - Traversal

Inserting- inserting integers into a binary search tree.

- **<u>Step 1 :</u>** If the current pointer is null. Create a new node, store the data , and return the address of the new node.
- Step 2 : otherwise compare the integer to the data stored at the current, if the new node is less than the integer at the current node, insert the new integer into the left child of the current node(by recursively applying the same algorithm), otherwise, insert it into the right child of the current node

Eg : 5,2,8,4 and 1



BST – Insert time complexity

The insert function requires time proportional to the height of the tree in the worst case. It takes `O(log n)` time to execute in the average case and `O(n)` time in the worst case.

Adding a new element to a binary search tree.

```
Input: binary search tree T, node v, element e
Output:
add(T, v, e){
     if(T.isLeaf(v)){
         if(v.element()>=e)
               add element e as v’s left child
         else
               add element e as v’s right child
   } else {
         if(v.element()>=e)
               add(T, T.leftChild(v), e)
         else
               add(T, T.rightChild(v), e)
     }
}
```

Adding a new element to a binary search tree.

```
Function insert(node, value):
    If node is NULL:
        Create a new node with the given value
        Return the new node  // This becomes the new subtree (leaf node)

    If value < node.value:
        node.left = insert(node.left, value)  //Recur/update the left child
    Else if value > node.value:
        node.right = insert(node.right, value)//Recur/update the right child
    Else:
        // Value already exists, do nothing (optional)

    Return node  // Pass the current node (or subtree root) back up
```

Adding a new element to a binary search tree.

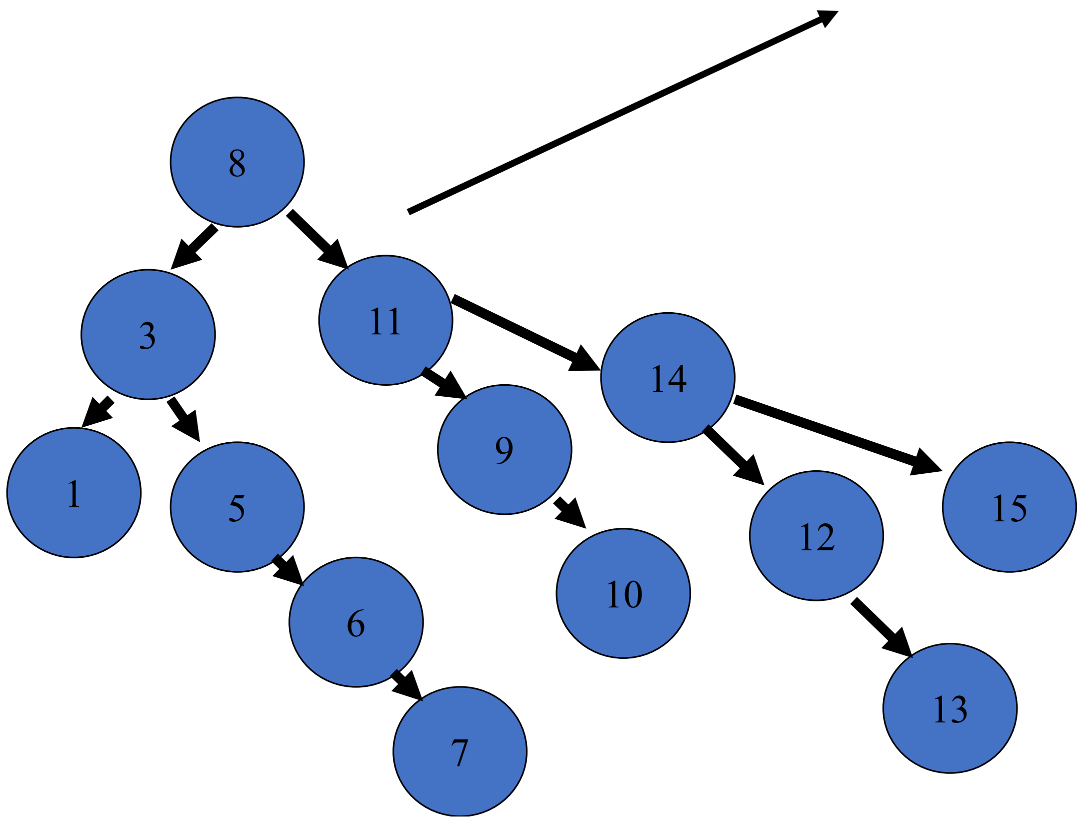
```
Input: binary search tree T, node v, element e
Output:
add(T, v, e){
     if(T.isLeaf(v)){
         if(v.element()>=e)
               add element e as v’s left child
         else
               add element e as v’s right child
   } else {
         if(v.element()>=e)
               add(T, T.leftChild(v), e)
         else
               add(T, T.rightChild(v), e)
     }
}
```

Deleting nodes from a binary search trees.

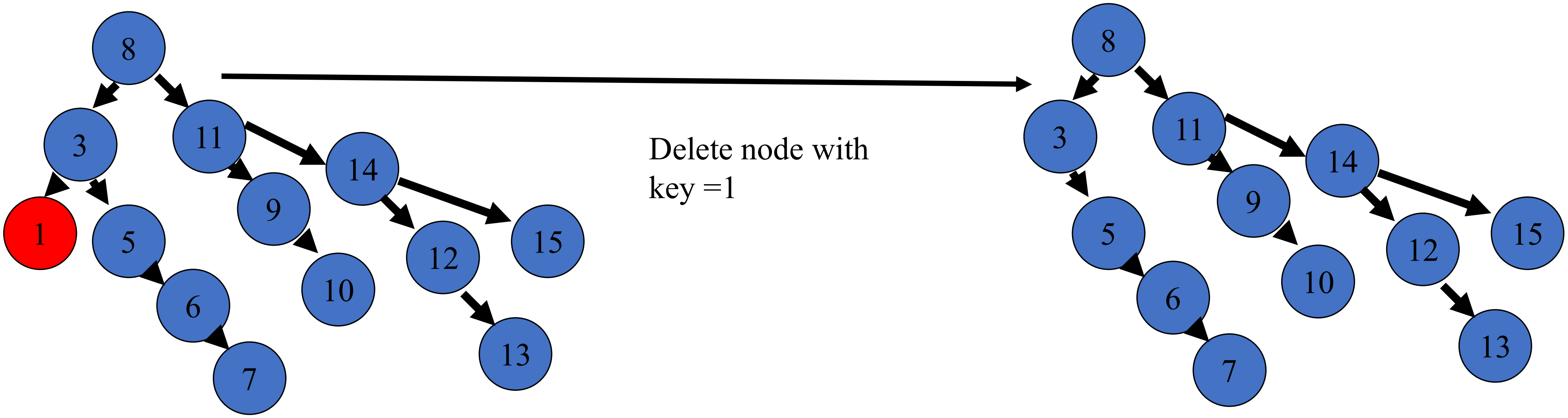
- There are three categories of nodes we may wish to delete.
- Category 1 : The node to be deleted has no sons.
  - The node can be deleted without any adjustment to the tree.
  - Delete the leaf and set the pointer from its parent (if any) to zero.

Example :Category 1

Delete node with key =1



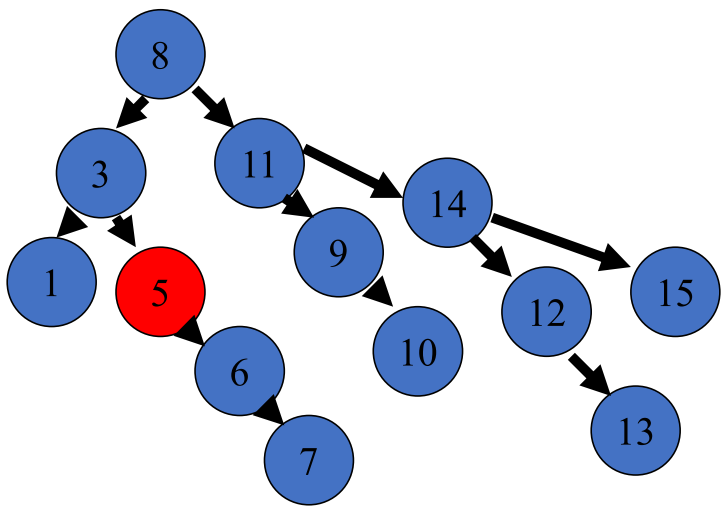
Example :Category 1



Category 2 : The node to be deleted has only one sub-tree.

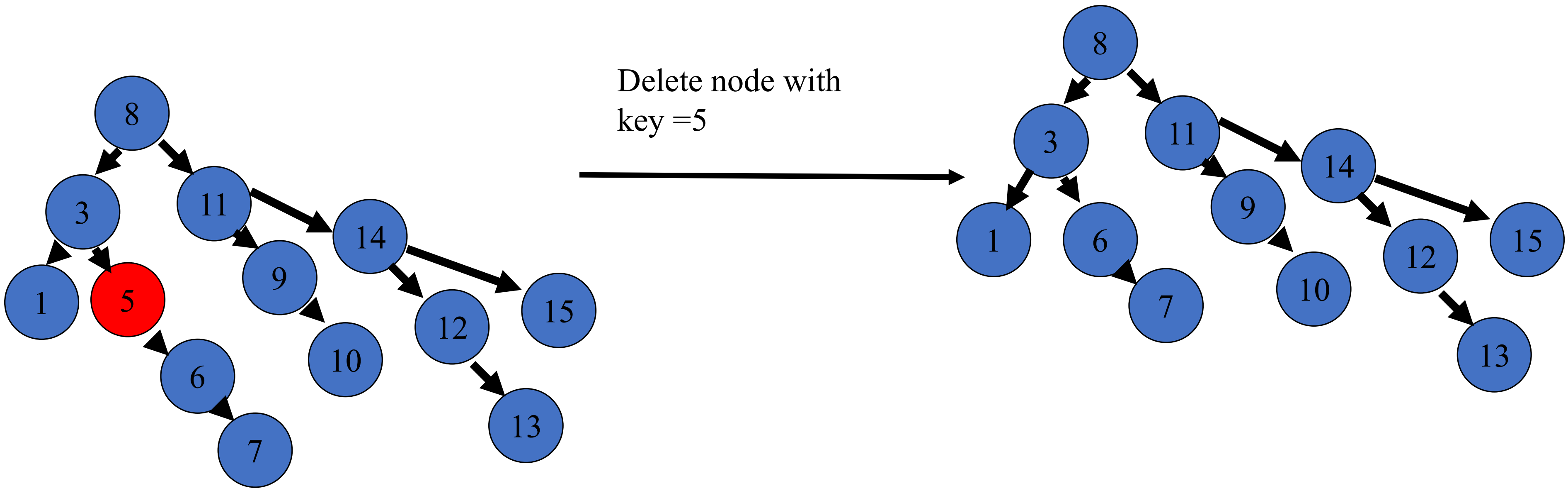
- Solution : Its only son can be moved up to its take place.
- We just redirect the references from the node’s parent so that it points to the child.

Example :Category 2



Delete node with
key =5

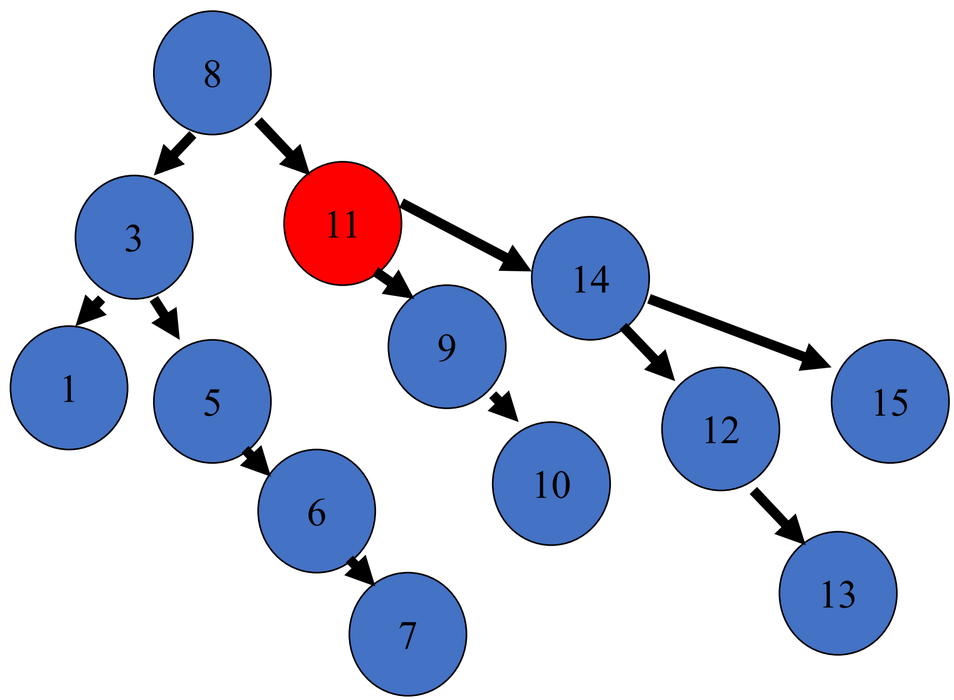
Example :Category 2



Category 3 : The node to be deleted has two sub-trees.

- Solution : this method we shall use is to replace the node being deleted by the rightmost node in its left sub-tree
- Or Leftmost mode in its right-sub-tree.

Example :Category 3

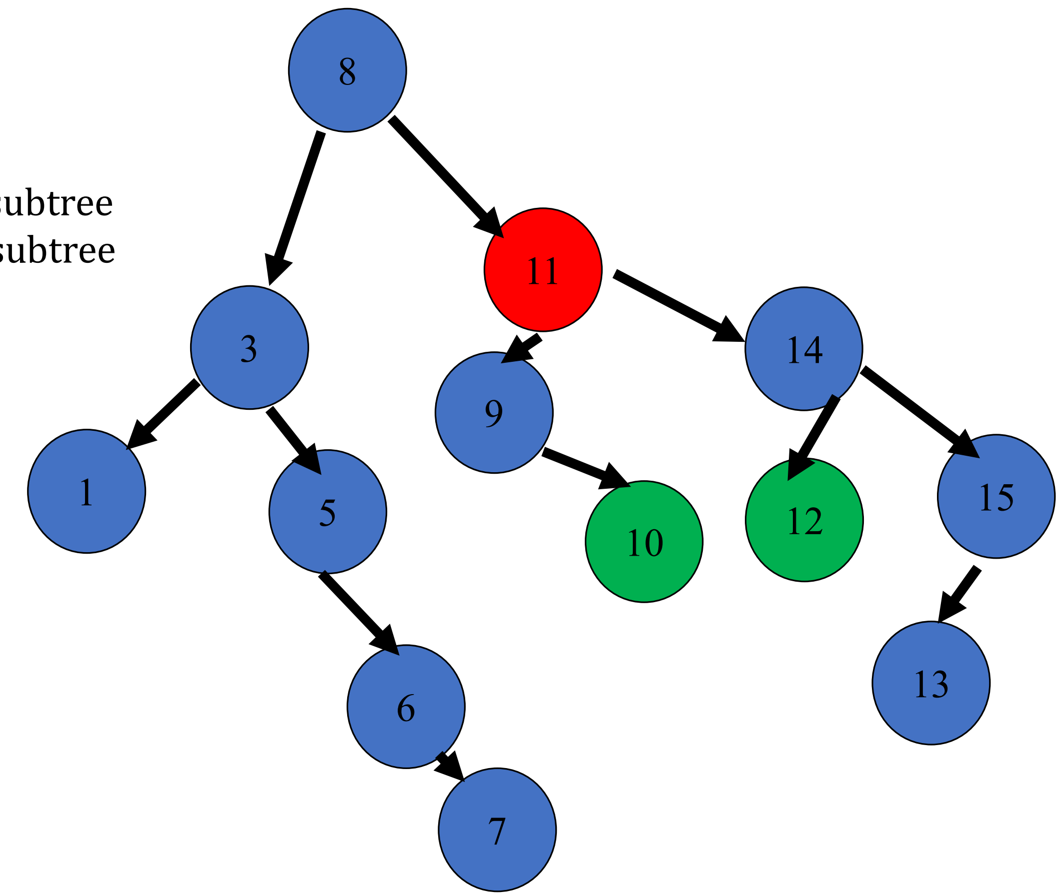


Delete node with
key =11

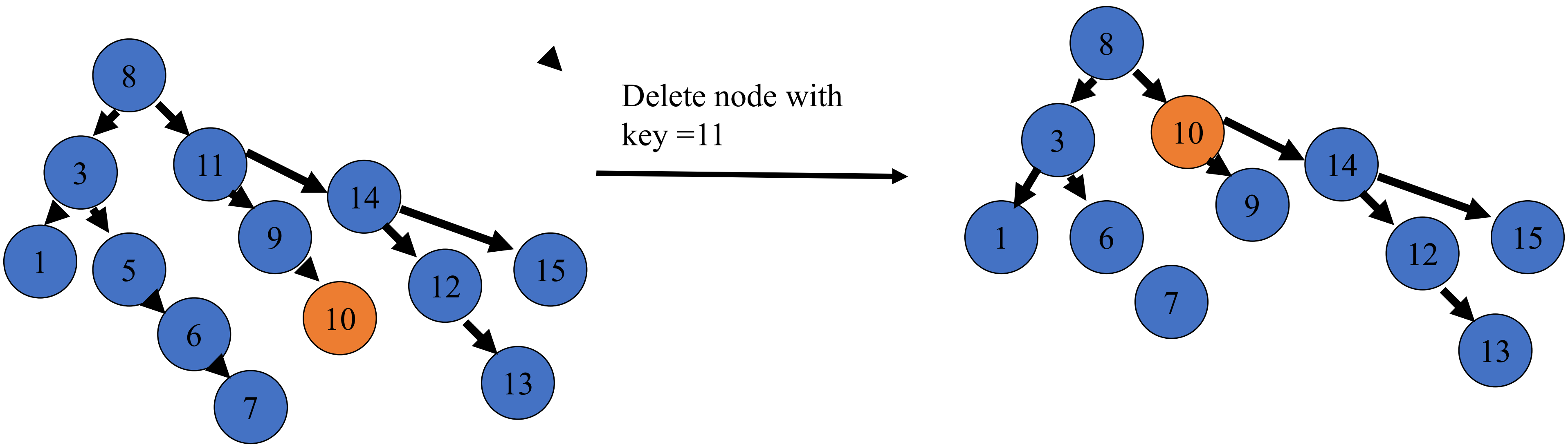
Example :Category 3

Suitable candidate to replace

1. Right most node of the left subtree
2. Left most node of the right subtree

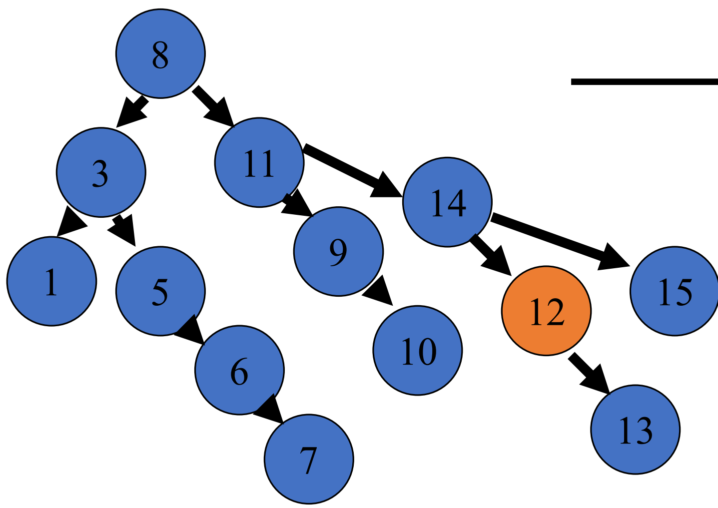


Example :Category 3 : using option 1

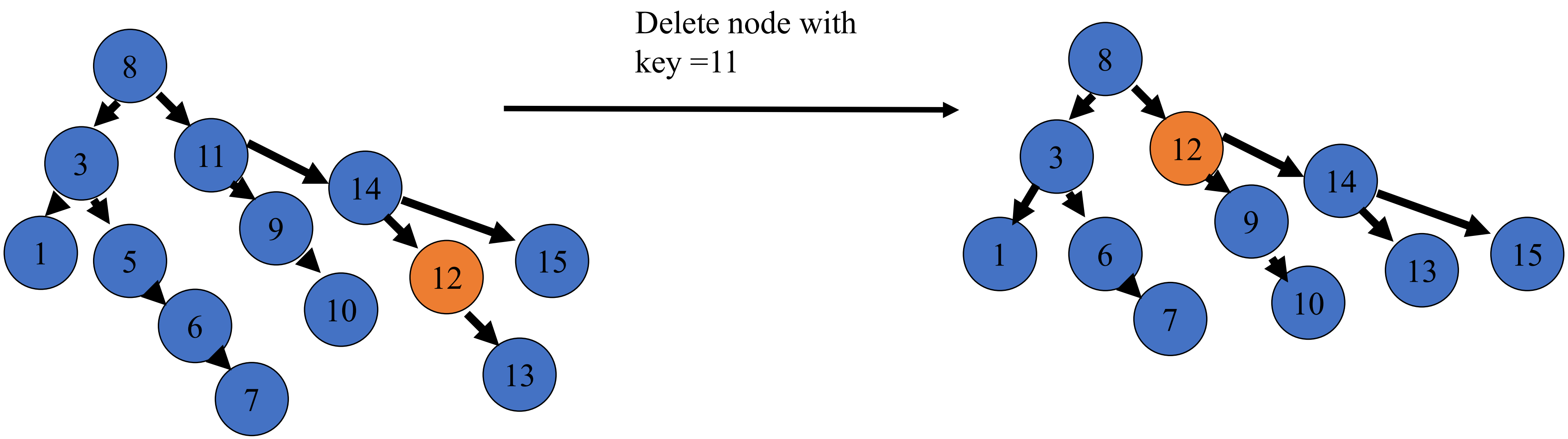


Example :Category 3 : : using option 2

Delete node with key =11



Example :Category 3



Algorithm to delete a node from a binary search tree

```
Delete (TREE, VAL)

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