Data Structures and Algorithms

Mr. Tahir Iqbal

tahir.iqbal@bahria.edu.pk

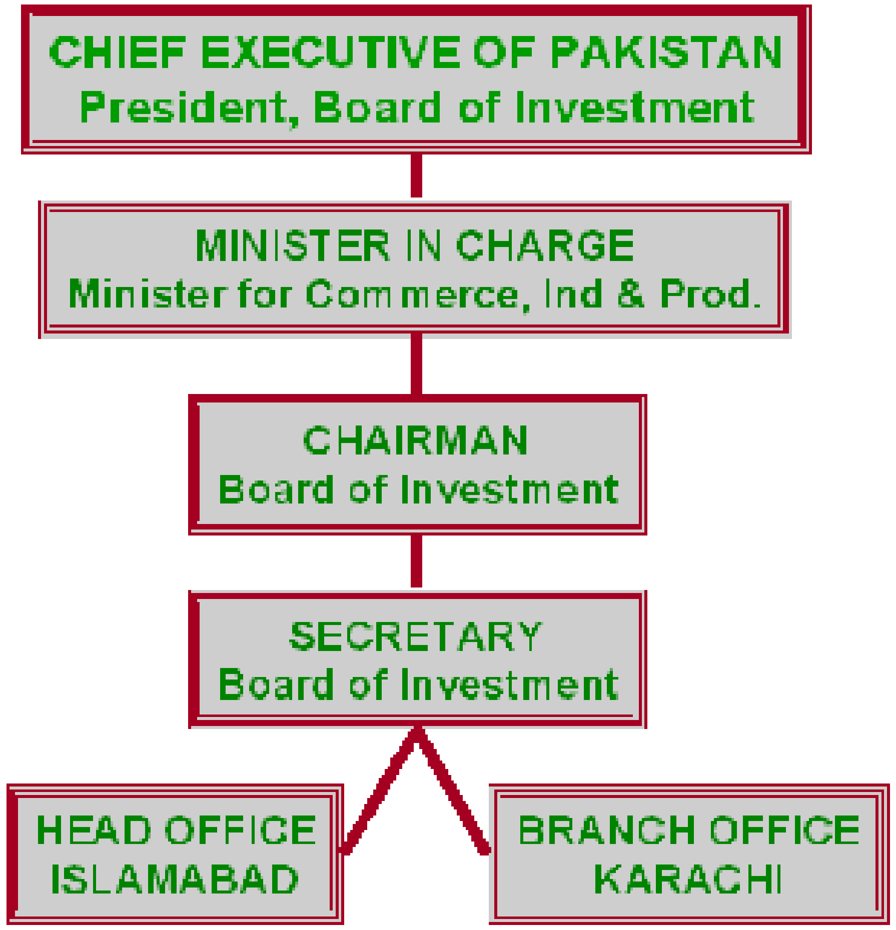
Lecture 11: Tree

Introduction

- Data structure such as Arrays, Stacks, Linked List and Queues are linear data structure. Elements are arranged in linear manner i.e. one after another.
- Tree is a non-linear data structure.
- Tree imposes a ***Hierarchical*** structure, on a collection of items.
- Several Practical applications
  - Organization charts.
  - Family hierarchy
  - Representation of algebraic expression

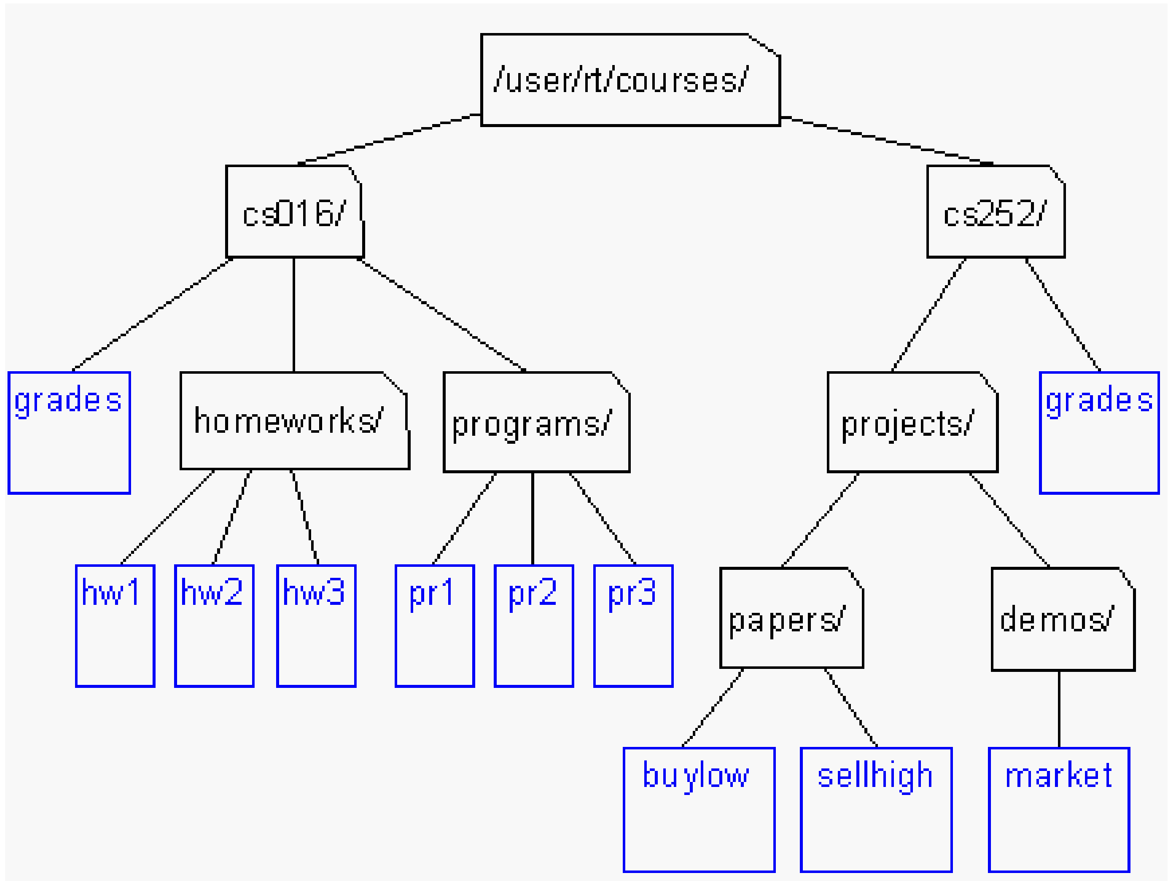
Example

- Organizational Chart



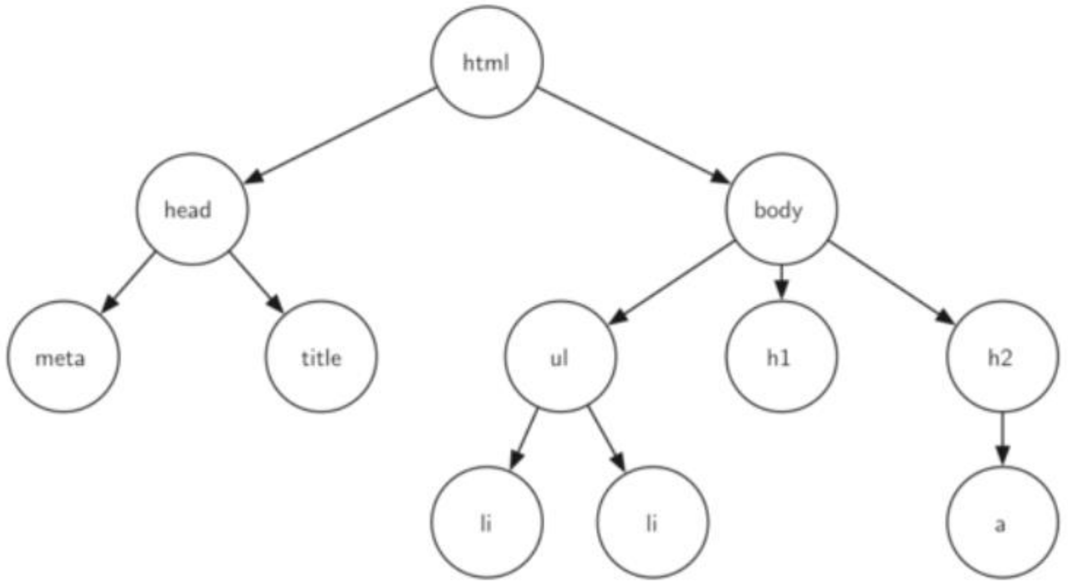
Example - Directory

Unix / Windows file structure



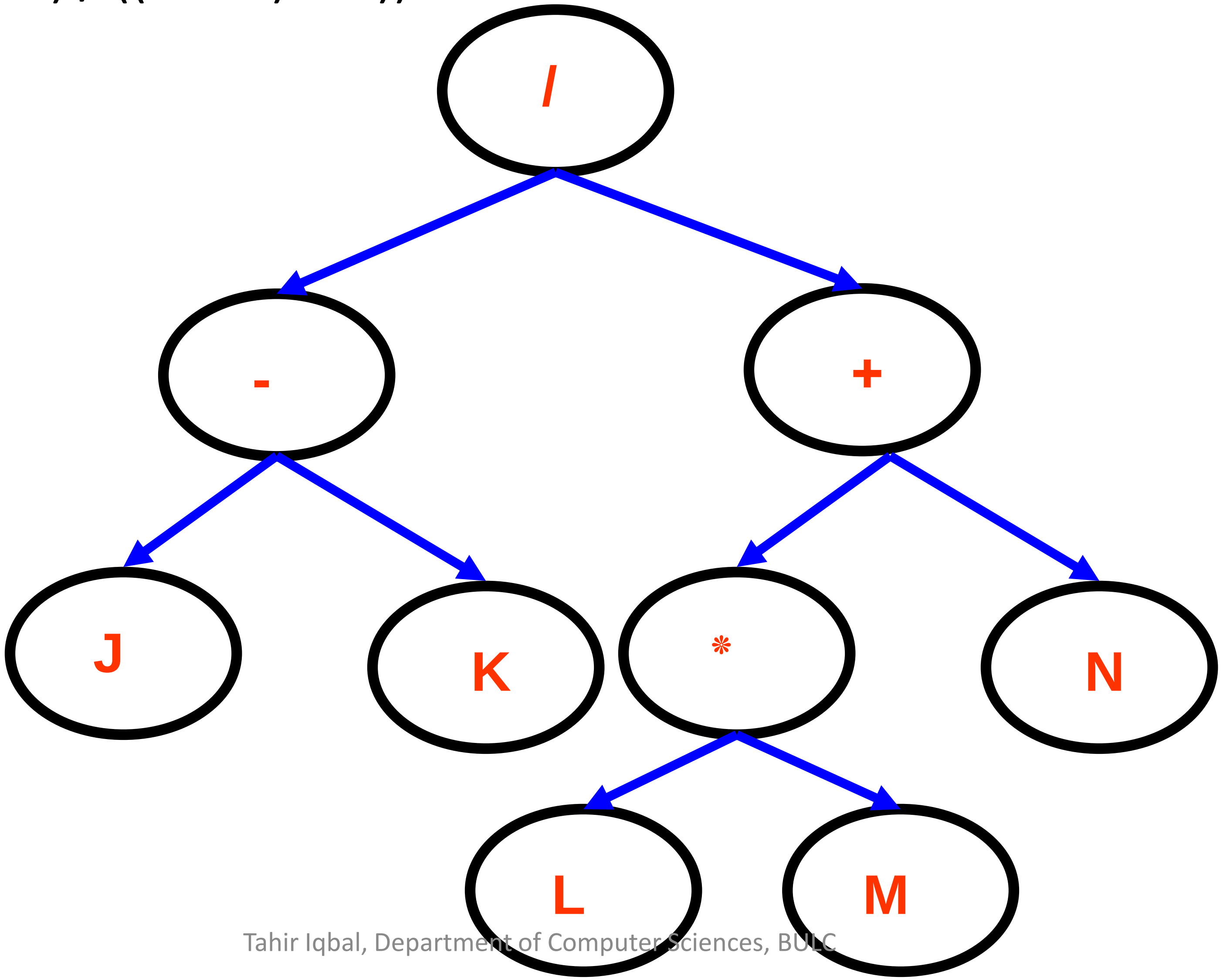
Example - Web

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en" lang="en">
<head>
    <meta http-equiv="Content-Type"
          content="text/html; charset=utf-8" />
    <title>simple</title>
</head>
<body>
<h1>A simple web page</h1>
<ul>
    <li>List item one</li>
    <li>List item two</li>
</ul>
<h2><a href="http://www.cs.luther.edu">
</body>
</html>
```



Example - Parse

- Representation of algebraic expression
- Z=((J - K) / ((L * M) + N))



Tree Definition

- A tree is a collection of nodes
  - The collection can be empty
  - If not empty, a tree consists of a distinguished node **R** (the *root*), and zero or more nonempty ***subtrees*** T_1, T_2, ..., T_k, each of whose roots are connected by a directed ***edge*** from **R.**

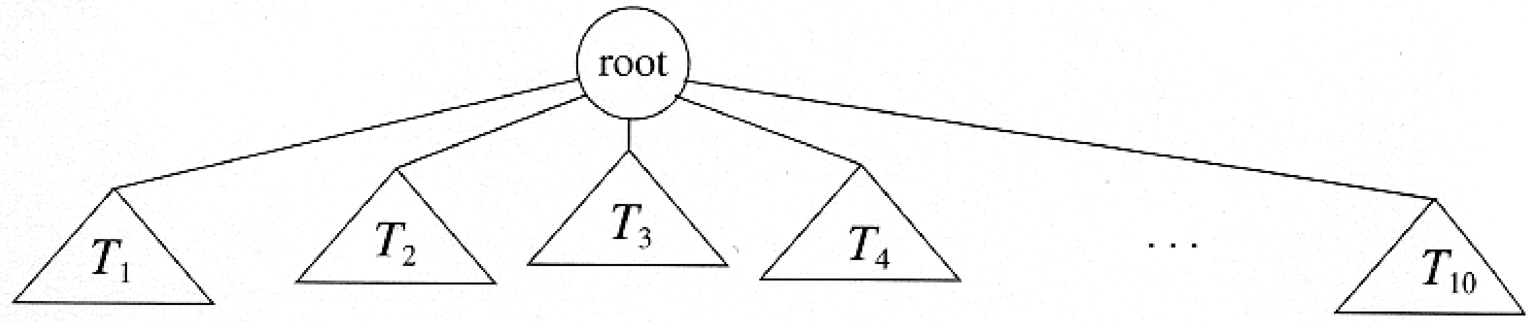


Figure 4.1 Generic tree

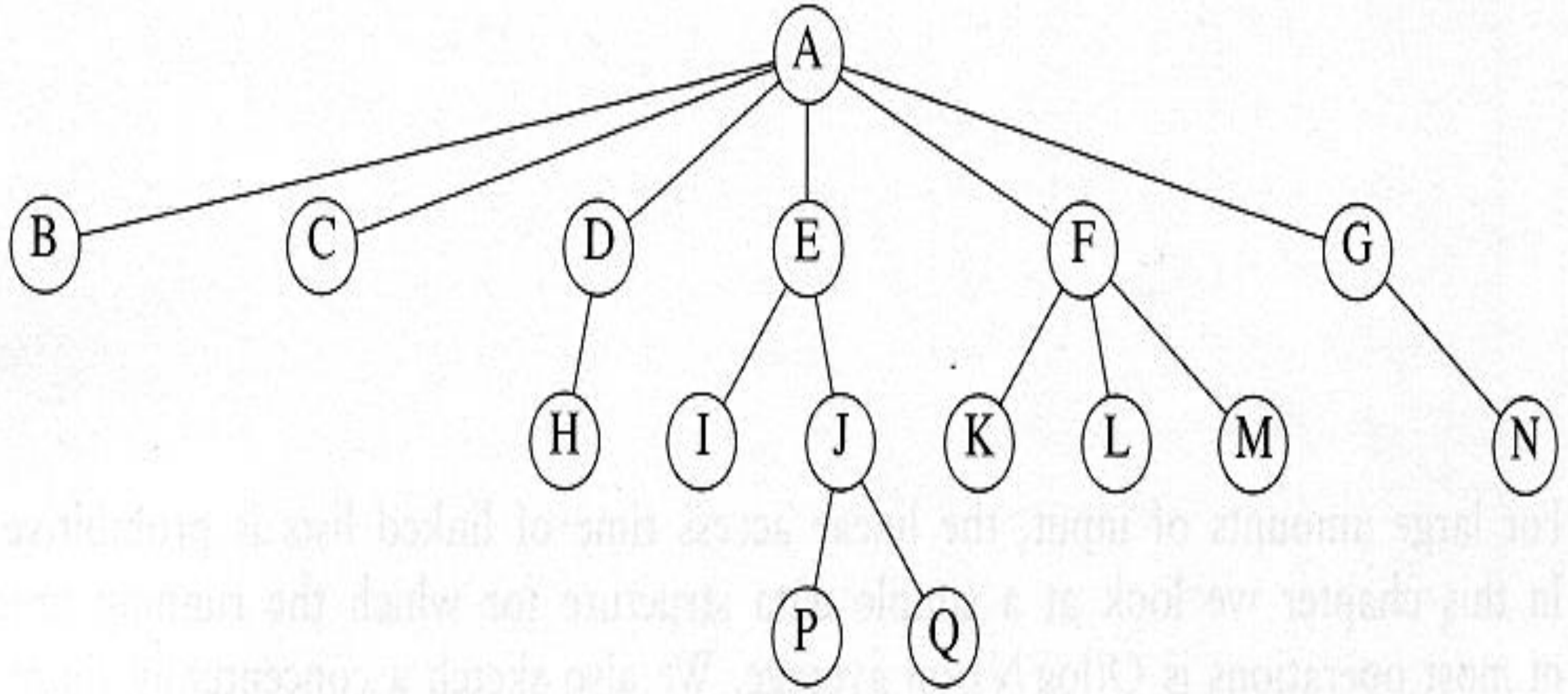
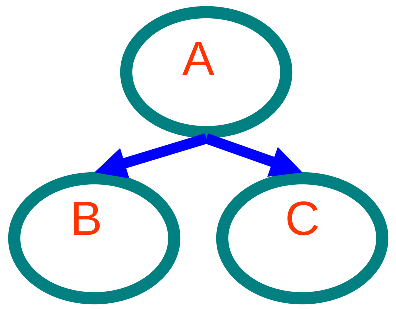


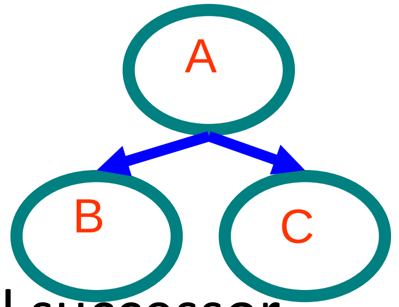
Figure 4.2 A tree

Tree Terminologies

- **Root**
  - It is the mother node of a tree structure. This tree does not have parent. It is the first node in hierarchical arrangement.
- **Node**
  - The node of a tree stores the data and its role is the same as in the linked list. Nodes are connected by the means of links with other nodes.
- **Parent**
  - It is the immediate predecessor of a node. In the figure A is the parent of B and C.



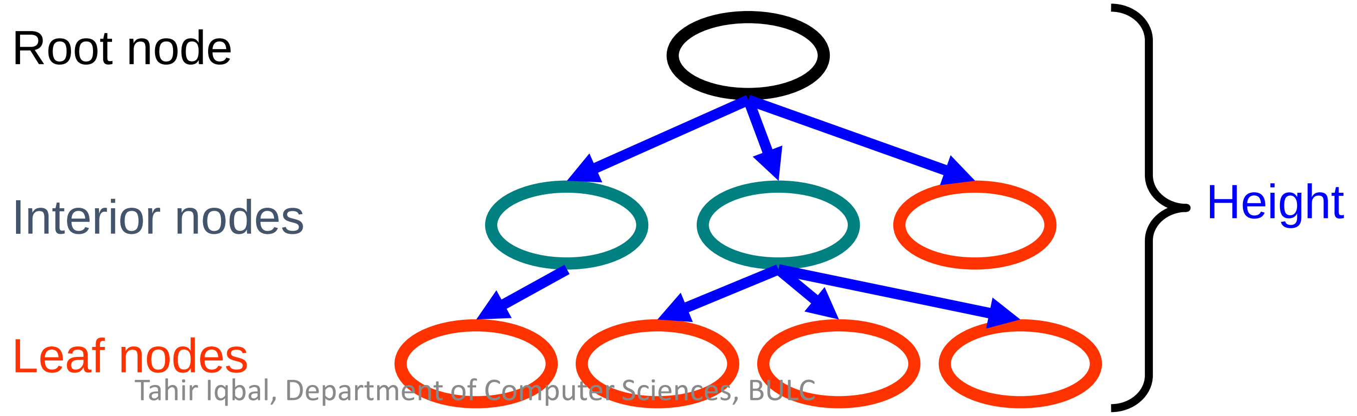
Tree Terminologies



- **Child**
  - When a predecessor of a node is parent then all successor nodes are called child nodes. In the figure B and C are the child nodes of A
- **Sibling**
  - The child node of same parent are called sibling. They are also called brother nodes.
- **Link / Edge**
  - An edge connects the two nodes. The line drawn from one node to other node is called edge / link. Link is nothing but a pointer to node in a tree structure.
- **Leaf / Terminal Node**
  - This node is located at the end of the tree. It does not have any child hence it is called leaf node.

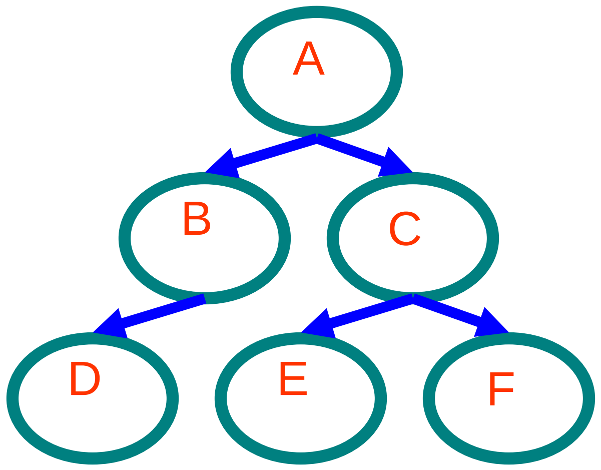
Tree Terminologies

- **Level**
  - Level is the rank of tree hierarchy. The whole tree structured is leveled. The level of the root node is always at 0. the immediate children of root are at level 1 and their children are at level 2 and so no.
- **Depth**
  - The **depth** of a node n is the length of the path from the root to the node. The set of all nodes at a given depth is sometimes called a **level** of the tree. The root node is at depth zero.
  - The **depth (or *height*)** of a tree is the length of the path from the root to the deepest node in the tree. A (rooted) tree with only one node (the root) has a depth of zero.



Tree Terminologies

- **Forest**
  - **It is a group of disjoint trees. If we remove a root node from a tree then it becomes the forest. In the following example, if we remove a root A then two disjoint sub-trees will be observed. They are left sub-tree B and right sub-tree C.**

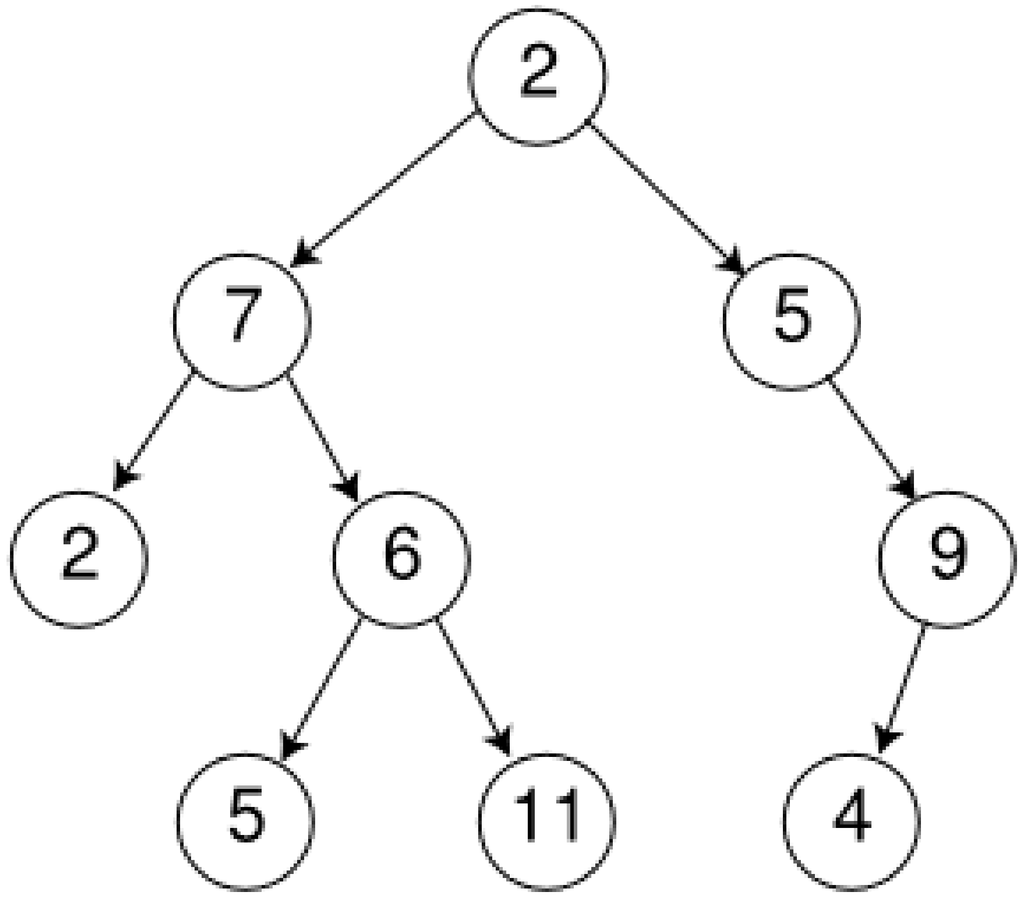


Binary Trees

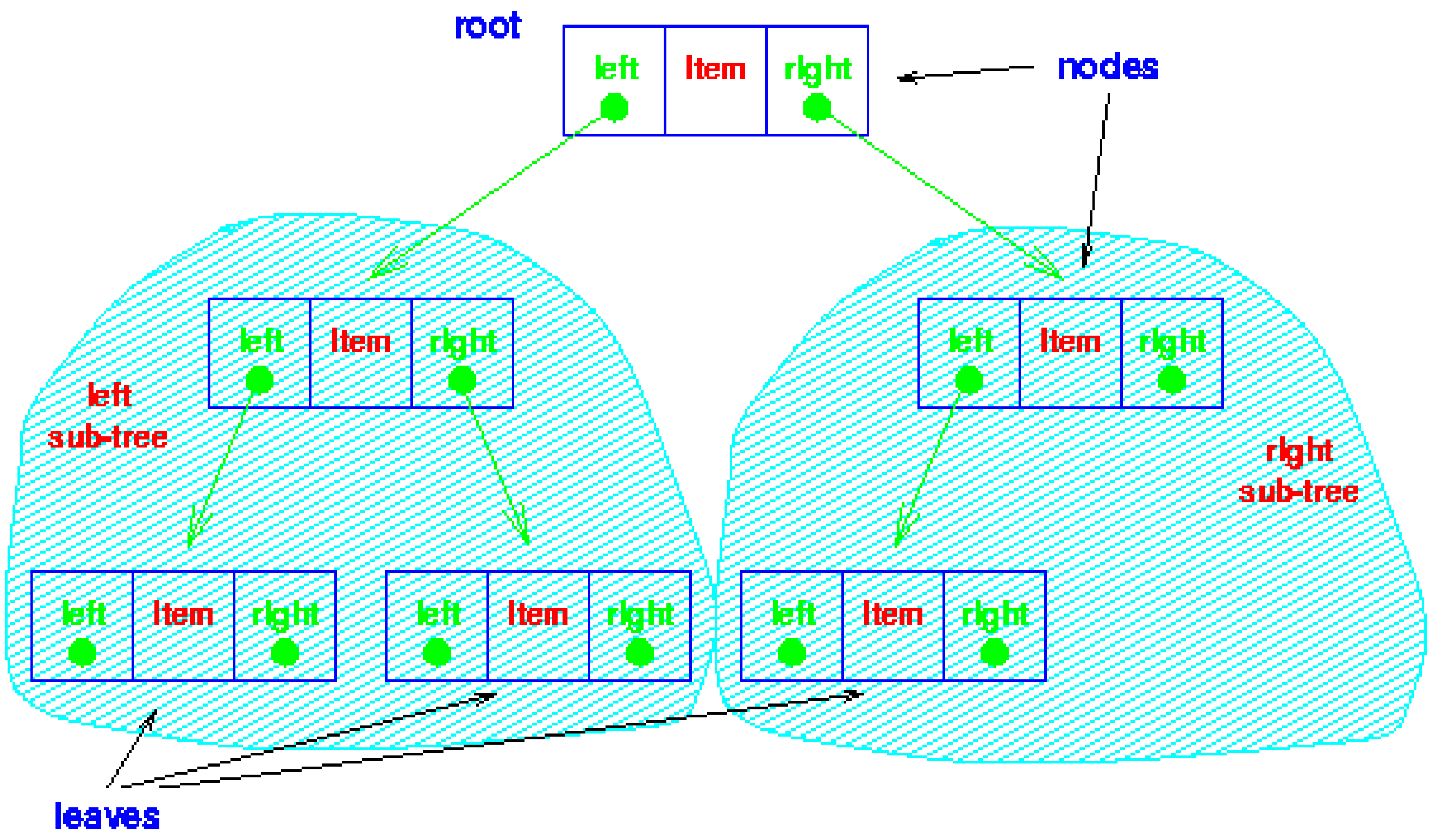
- The simplest form of tree is a **binary tree**. A binary tree consists of
  - a *node* (called the **root** node) and
  - left and right sub-trees.
    Both the sub-trees are themselves binary trees

- Also, a tree is binary if each node of it has a maximum of two branches i.e. a node of a binary tree can have maximum two children.

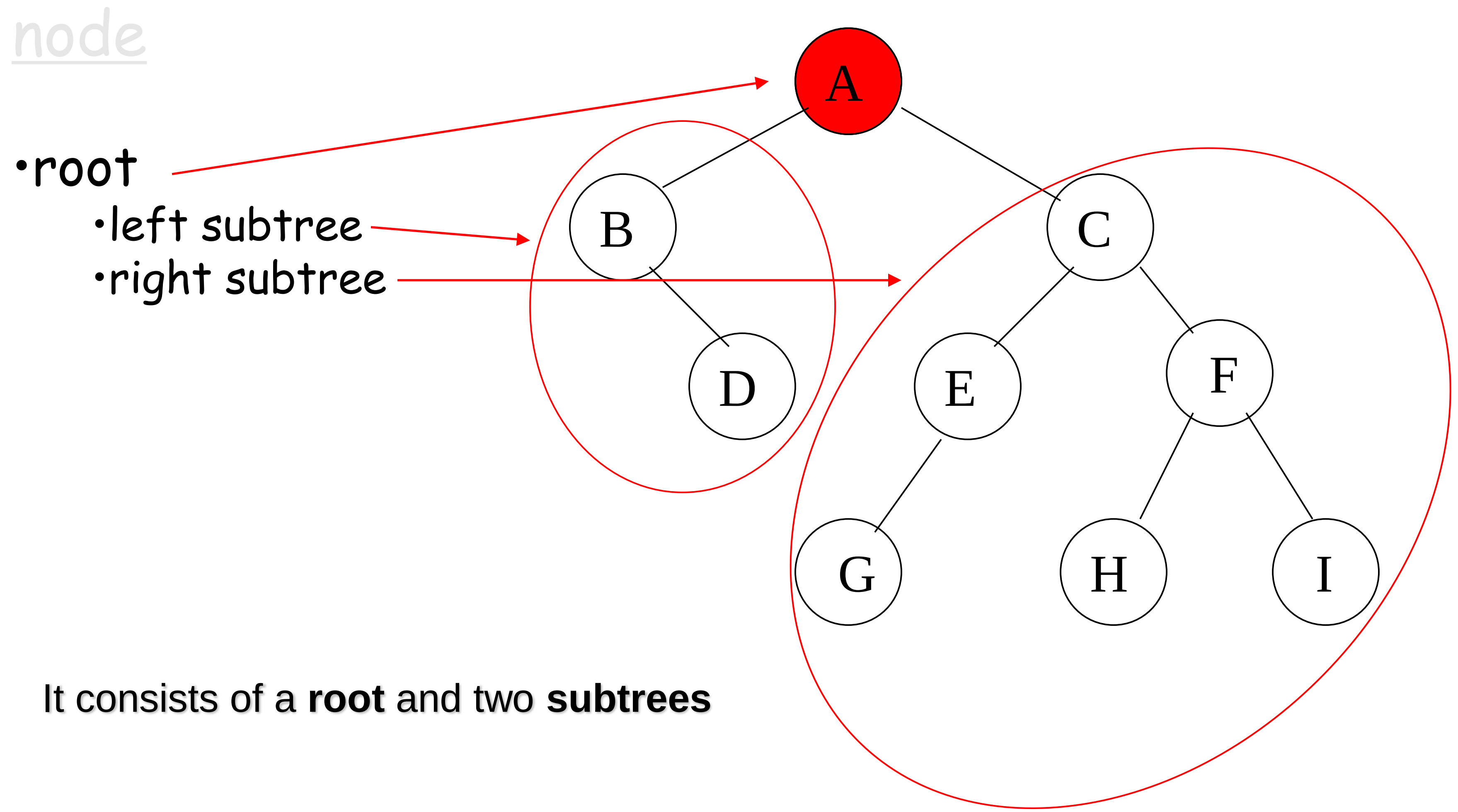
A Binary Tree



Binary Tree

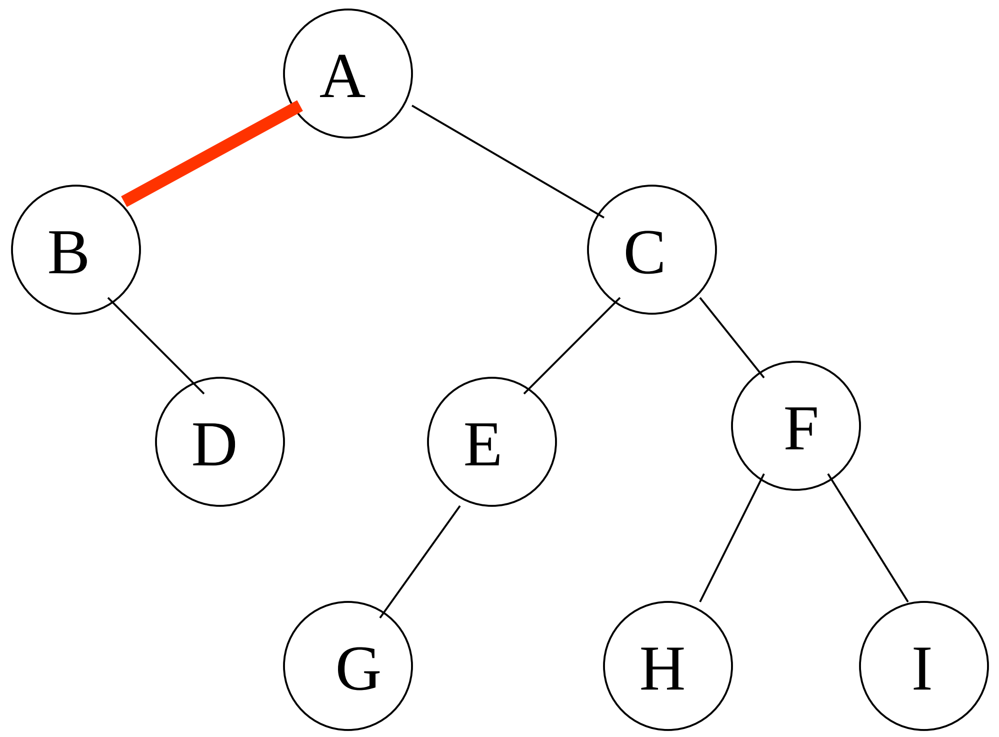

root
left
Item
right
nodes
left
sub-tree
right
sub-tree
leaves

Notation



It consists of a **root** and two **subtrees**

Notation

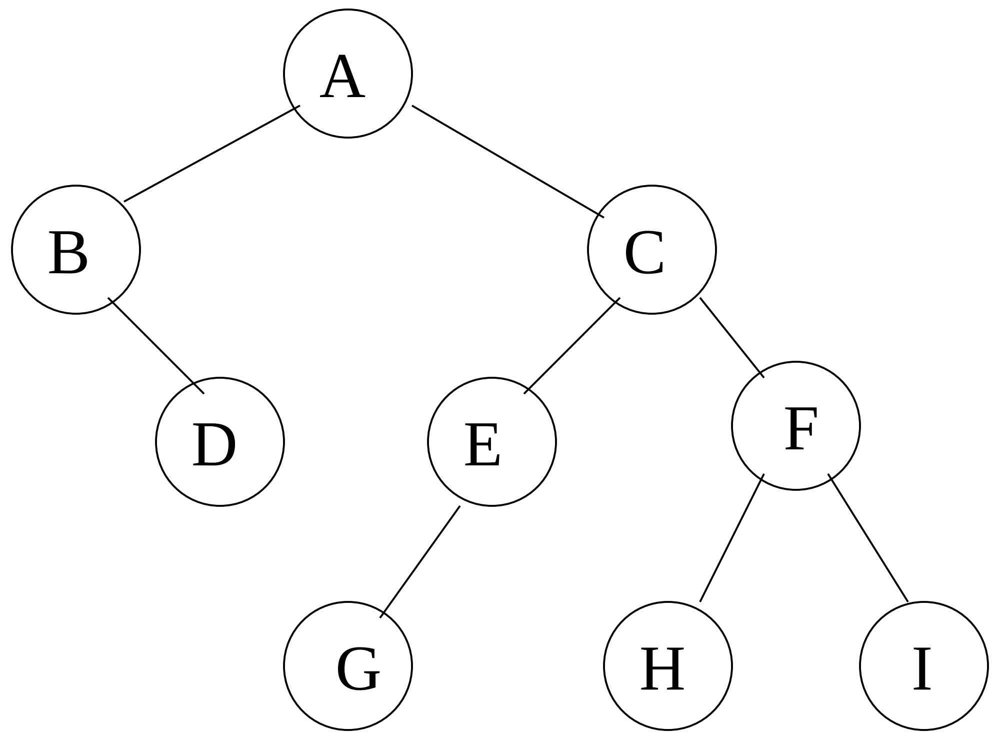


edge -

there is an **edge** from the root to its children

Notation

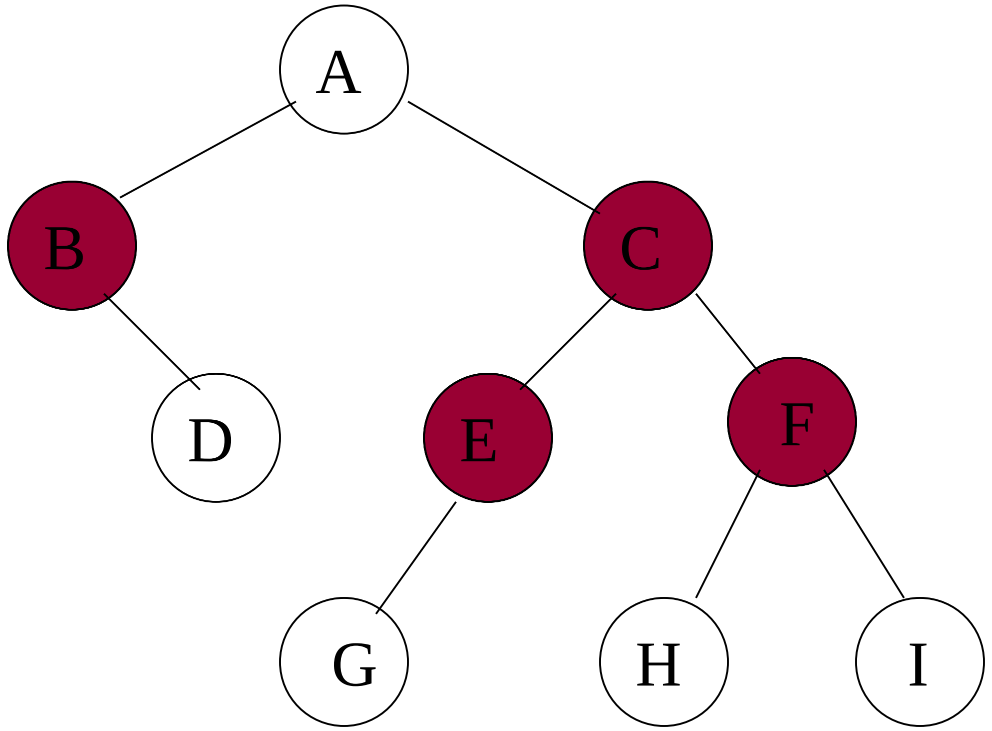
Child nodes



Notation

children

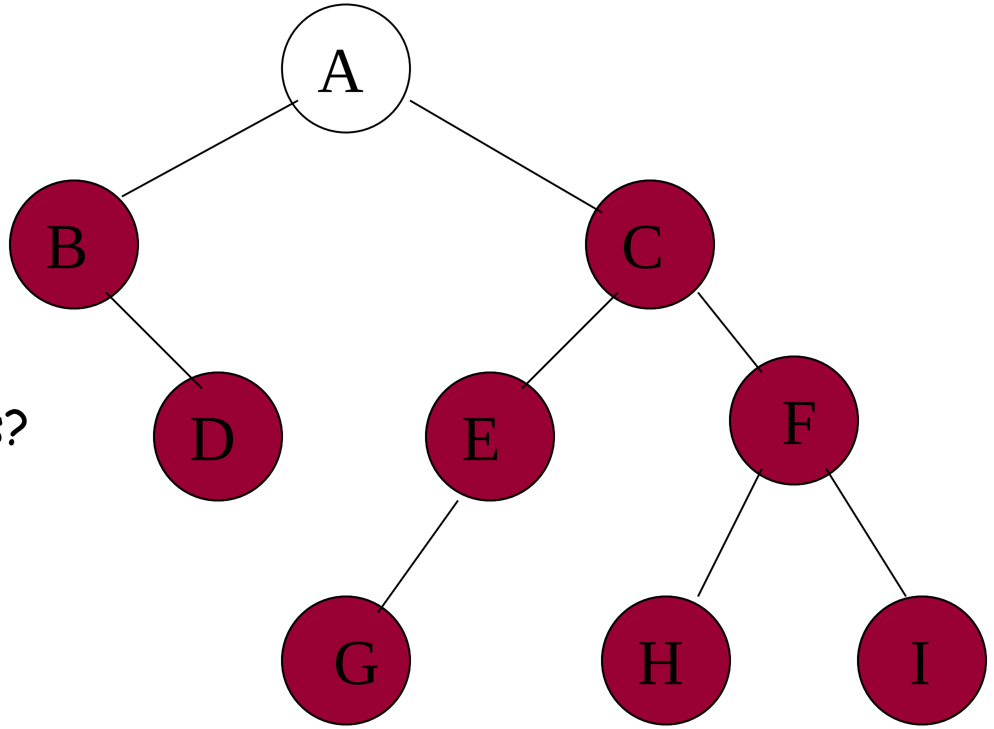
Child nodes of C?



Child nodes of A

Notation

descendants



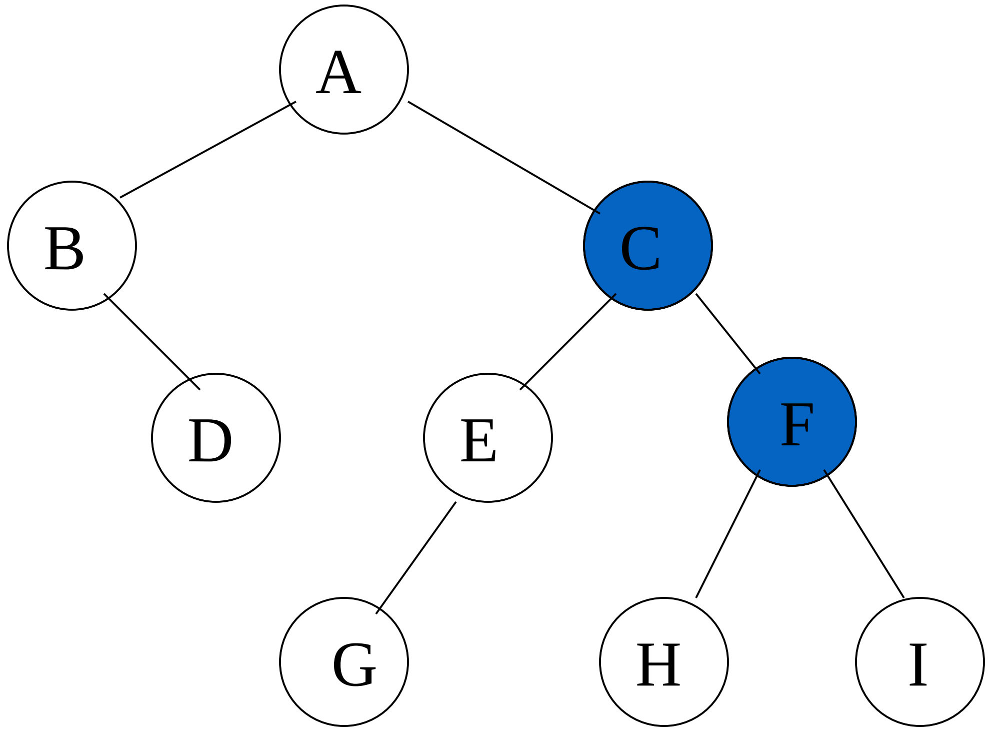
Who are node C's descendants?

Who are node A's descendants?

Notation

parents

Who is node E's parent?

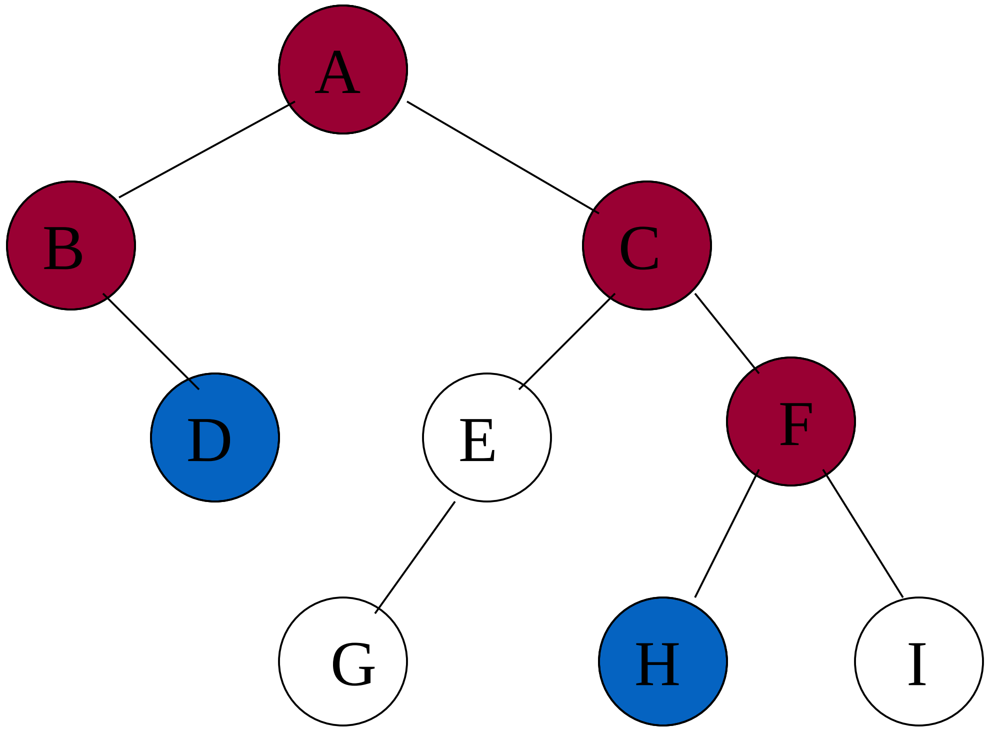


Who is node H's parent?

Notation

ancestors

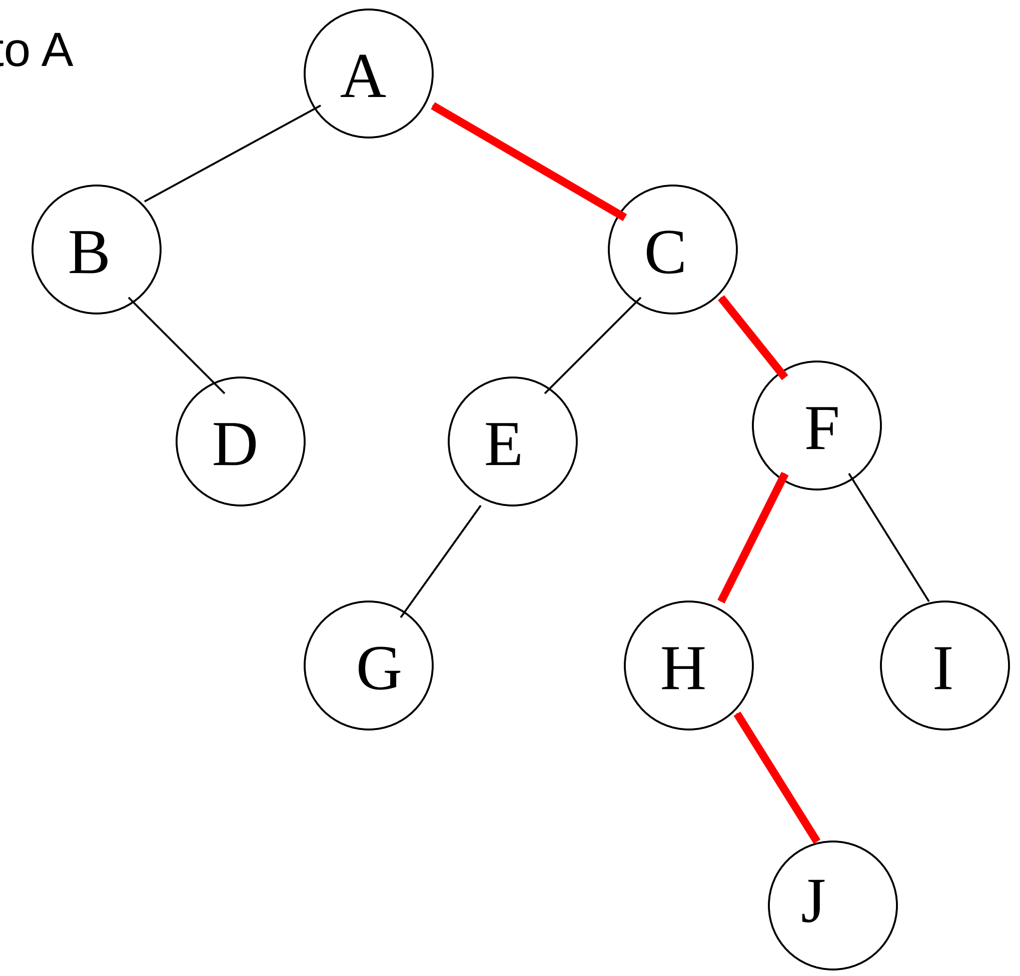
Who are node D's ancestors?



Who are node H's ancestors?

Notation

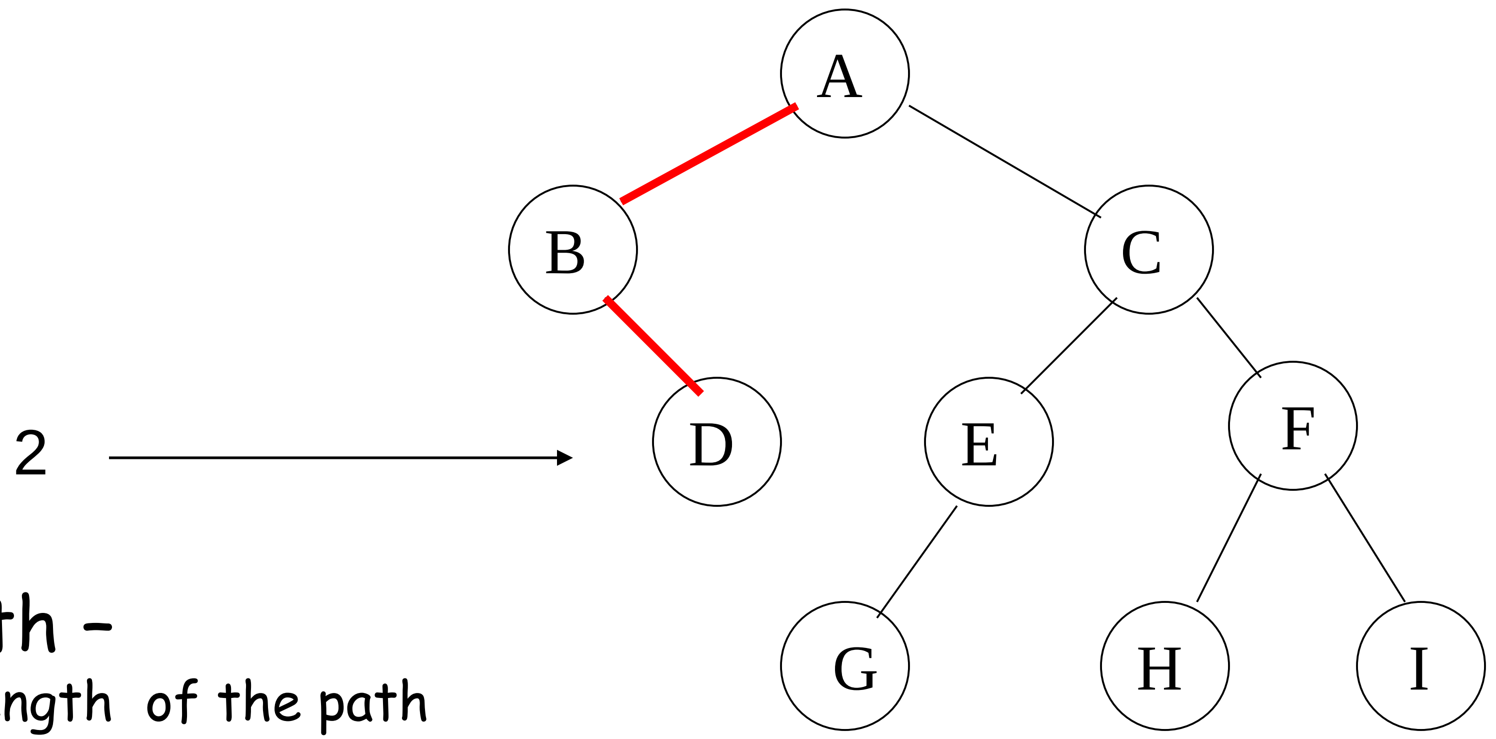
from J to A



path –

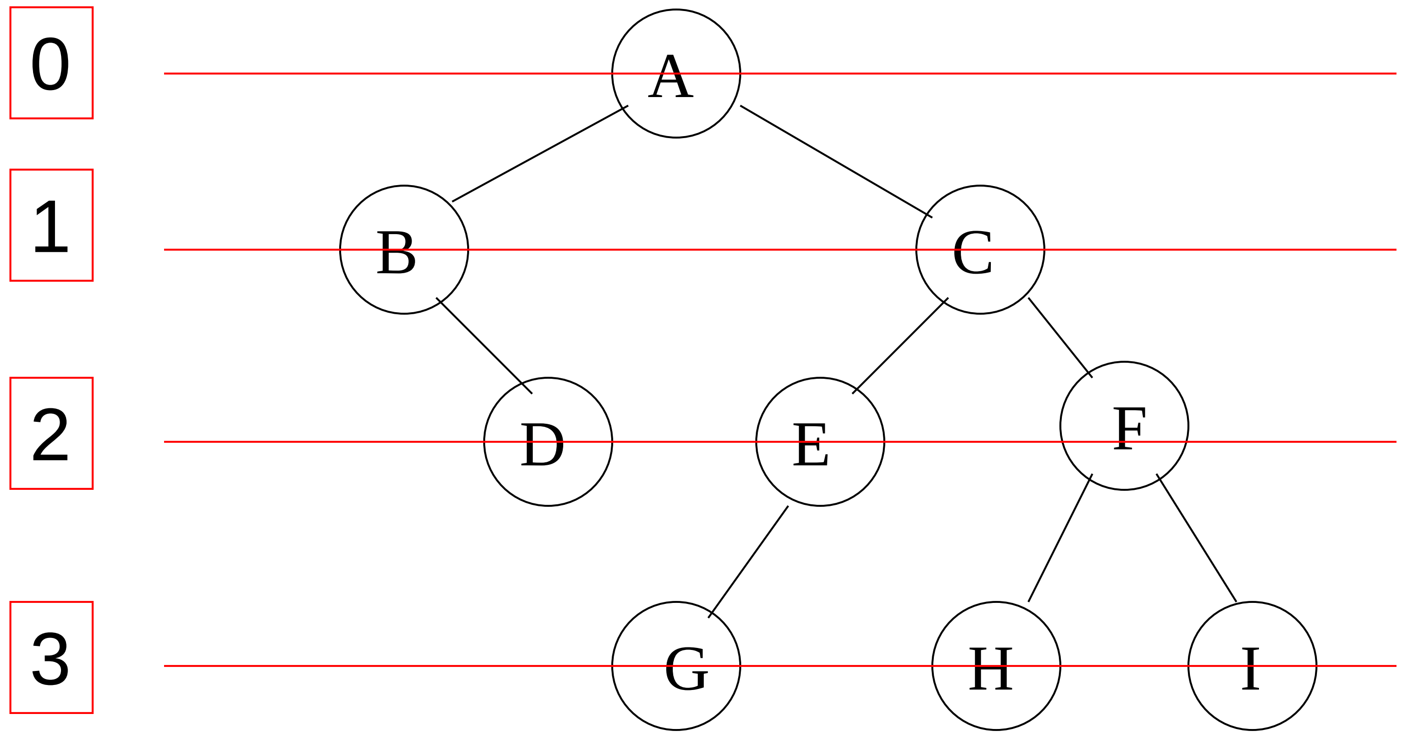
If n1, n2,…nk is a sequence of nodes such that ni is the parent of ni+1, then that sequence is a **path.**

Notation



depth –
the length of the path from the root of the tree to the node

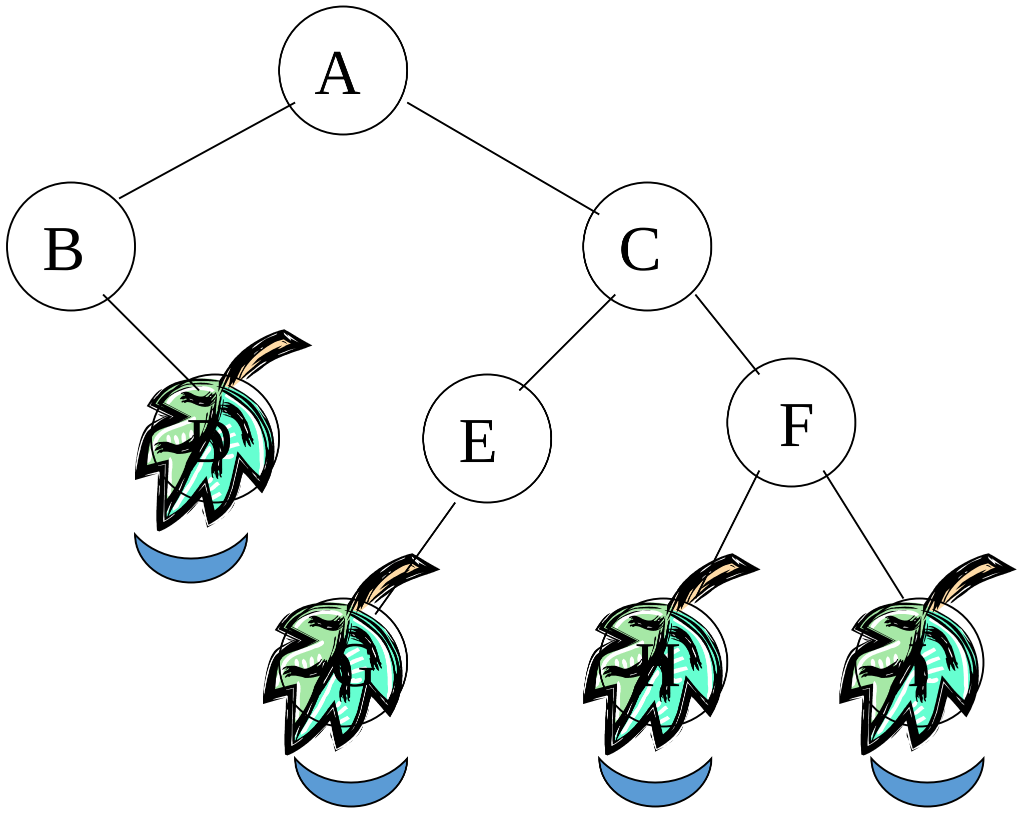
Notation



level –

all nodes of depth d are at level d in the tree

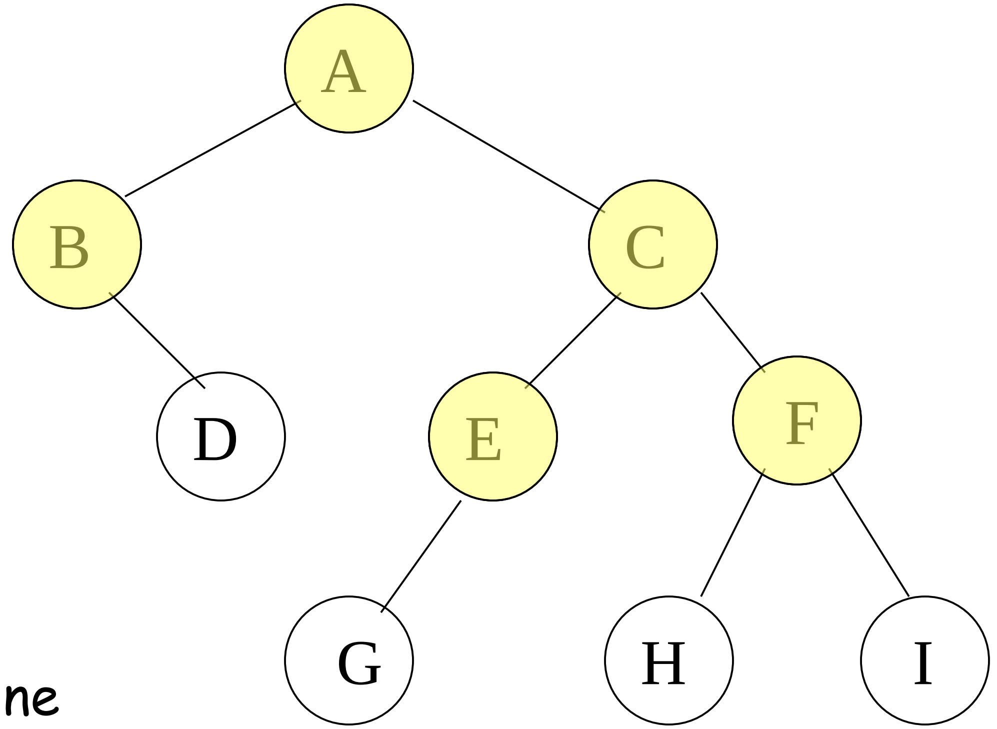
Notation



leaf node -

any node that has two empty children

Notation



internal node –
any node that has at least one non-emptyChild

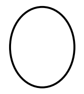
Or

An *internal node* of a *tree* is any *node* which has degree greater than one.

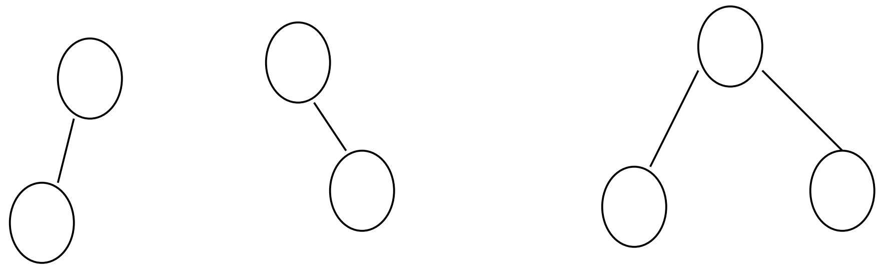
Binary Trees

Some Binary Trees

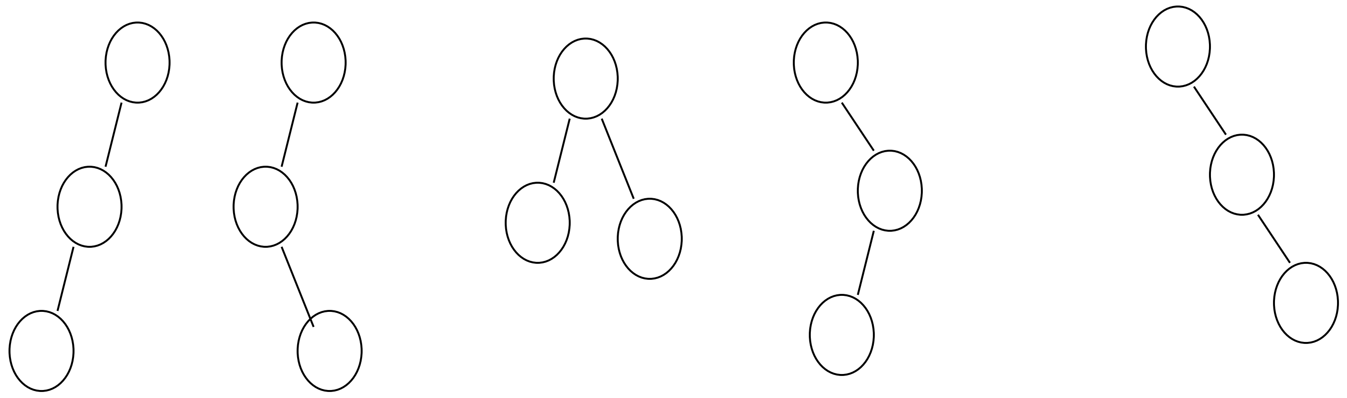
One node



Two nodes

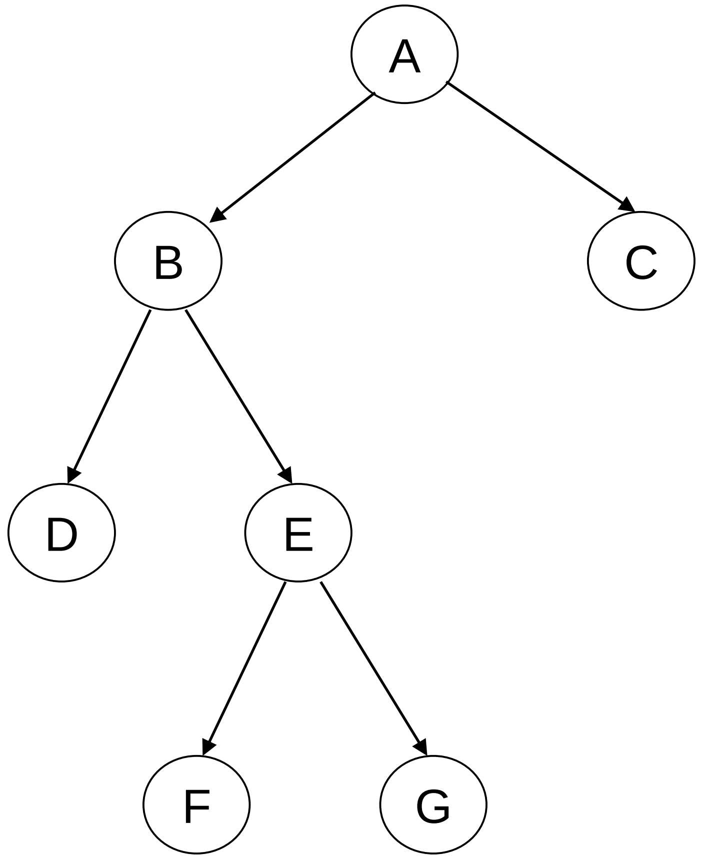


Three nodes



Strictly Binary Tree

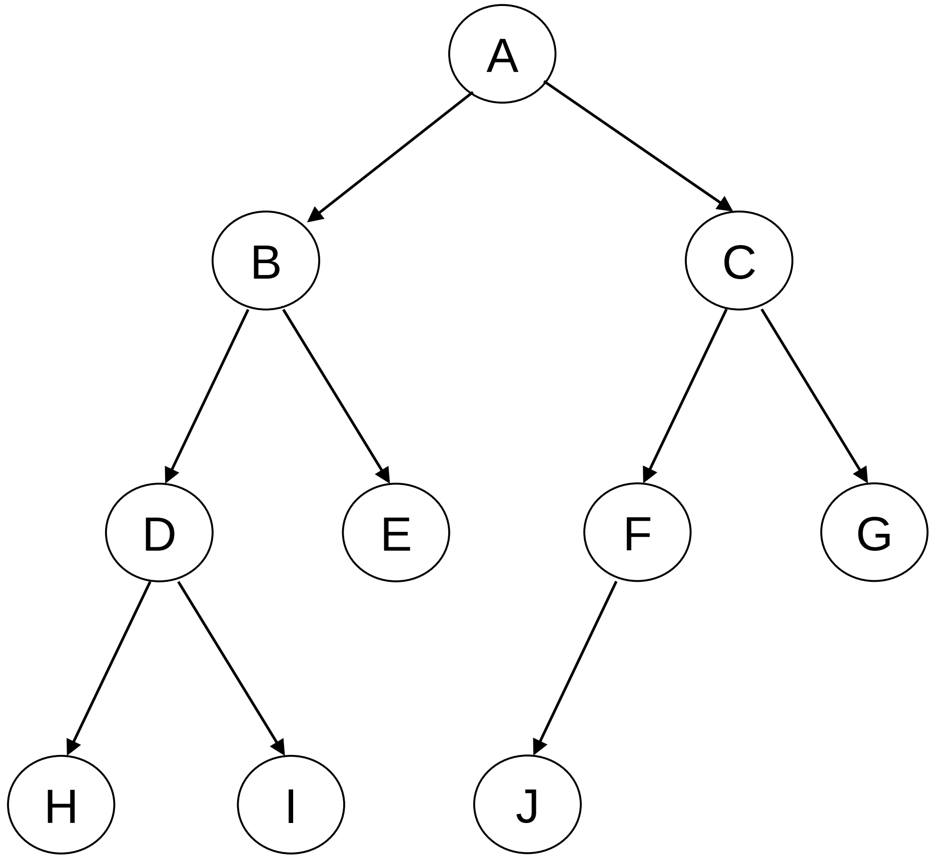
- When every non-leaf node in binary tree is filled with left and right sub-trees, the tree is called strictly binary tree.



Complete Binary Tree

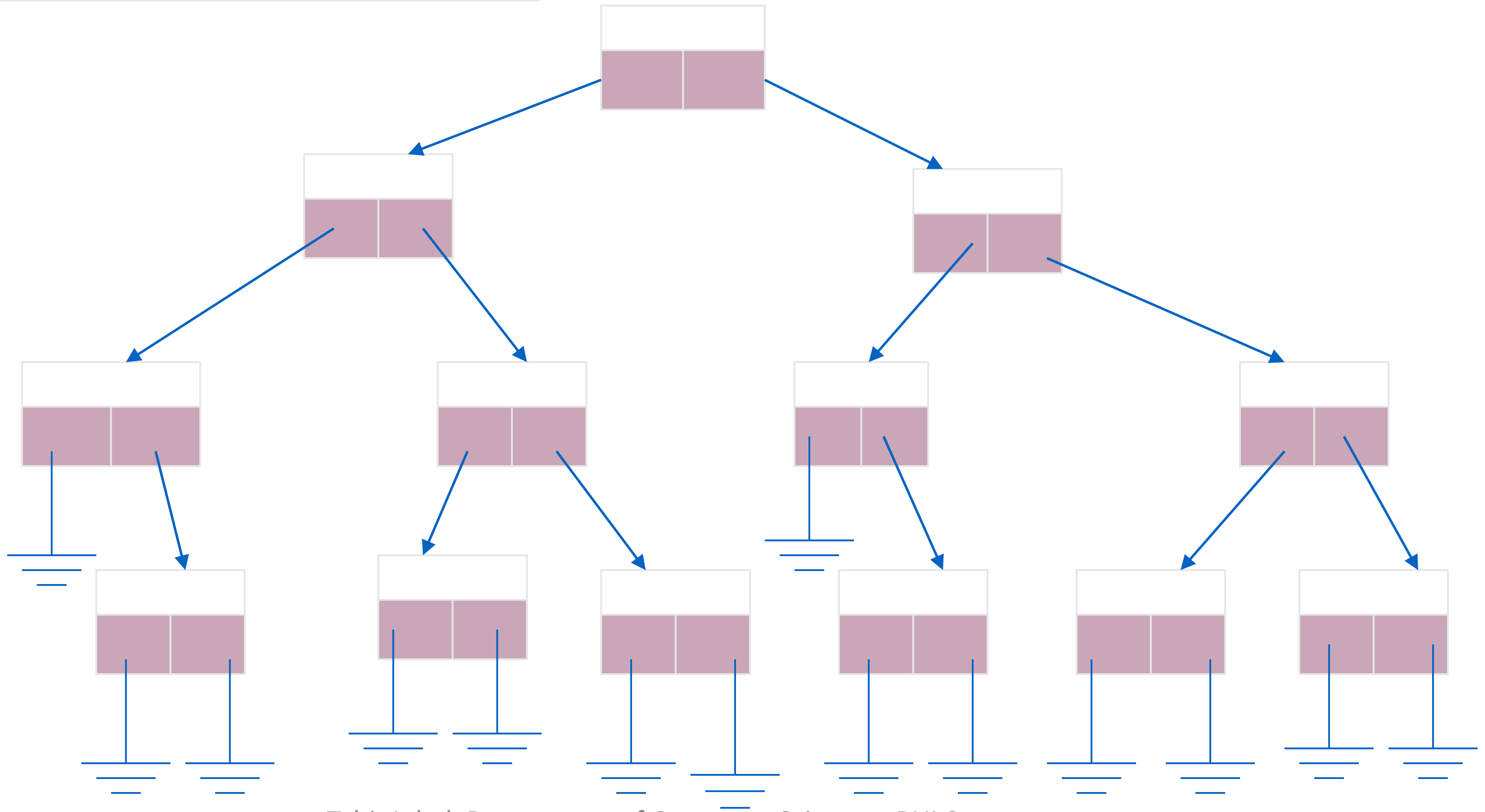
A ***Complete binary*** tree is:

- A tree in each level of the tree is completely filled.
- Except, possibly the bottom level.



Dynamic Implementation of Binary Tree

Linked Implementation



Structure Definition of Binary Tree Using Dynamic Implementation

- The fundamental component of binary tree is node.
- In binary tree node should consist of three things.
  - **Data**
    - **Stores given values**
  - **Left child**
    - **is a link field and hold the address of its left node**
  - **Right child.**
    - **Is a link field and holds the address of its right node**.

```
struct node
{
        int data
        node *left_child;
        node *right_child;
};
```

Operations on Binary Tree

- **Create**
  - **Create an empty binary tree**
- **Empty**
  - **Return true when binary tree is empty else return false.**
- **Tree Traversal**
  - **Inorder Traversal**
  - **Preorder Traversal**
  - **Postorder Traversal**
- **Insert**
  - **To insert a node**
- **Deletion**
  - **To delete a node**
- **Search**
  - **To search a given node**
- **Copy**
  - **Copy one tree into another.**

Applications of binary trees

Binary Search Tree - Used in *many* search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.

Binary Space Partition - Used in almost every 3D video game to determine what objects need to be rendered.

Binary Tries - Used in almost every high-bandwidth router for storing router-tables.

Hash Trees - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.

Heaps - Used in implementing efficient priority-queues, which in turn are used for scheduling processes in many operating systems, Quality-of-Service in routers, and A* *(path-finding algorithm used in AI applications, including robotics and video games)*. Also used in heap-sort.

Huffman Coding Tree (Chip Uni) - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.

GGM Trees - Used in cryptographic applications to generate a tree of pseudo-random numbers.

Syntax Tree - Constructed by compilers and (implicitly) calculators to parse expressions.

Treap - Randomized data structure used in wireless networking and memory allocation.

T-tree - Though most databases use some form of B-tree to store data on the drive, databases which keep all (most) their data in memory often use T-trees to do so.

Traversals

Example: Expression Trees

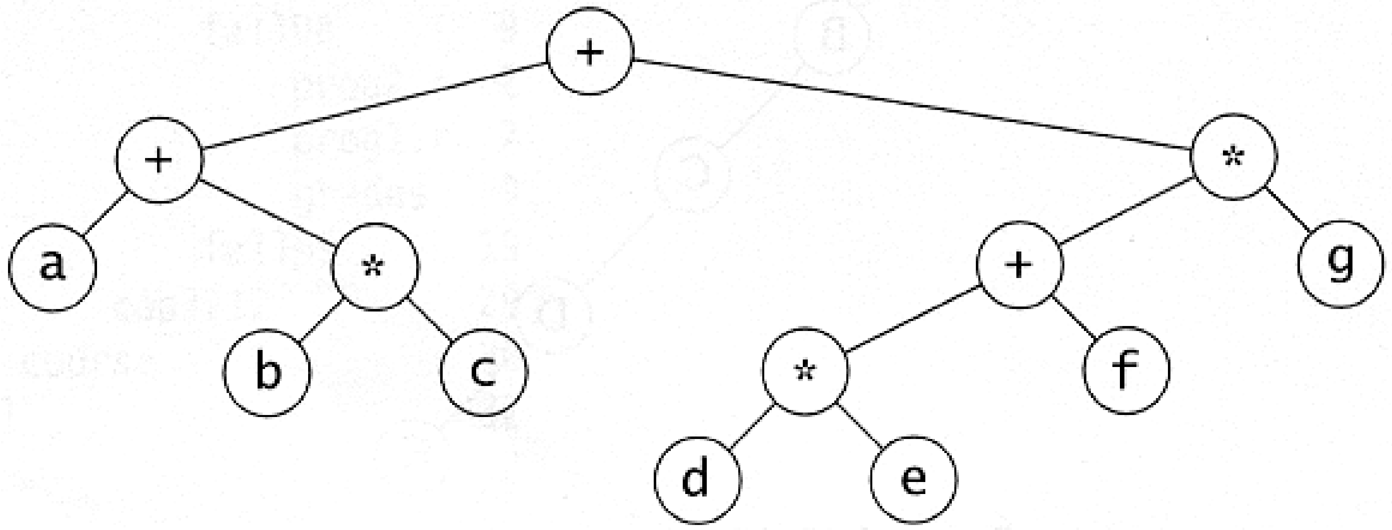
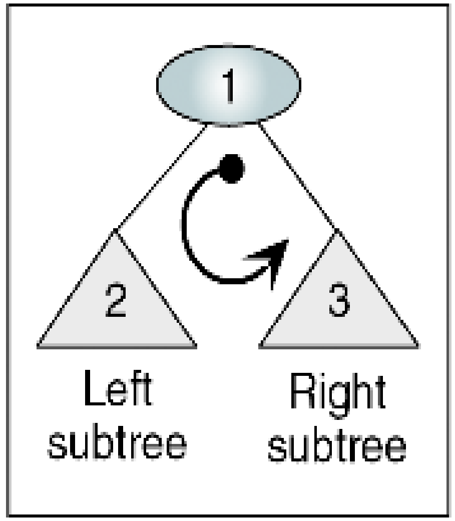


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

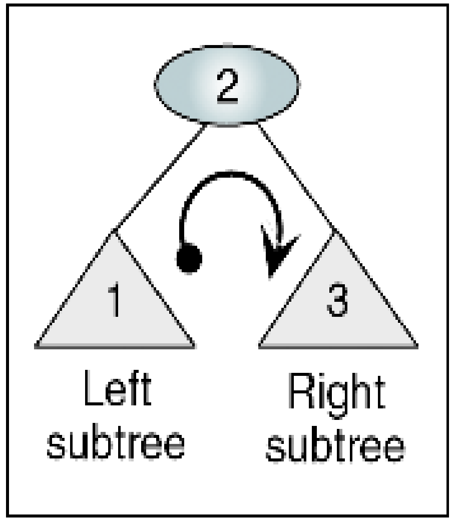
- Leaves are operands (constants or variables)
- The other nodes (internal nodes) contain operators

Traversal of a Binary Tree

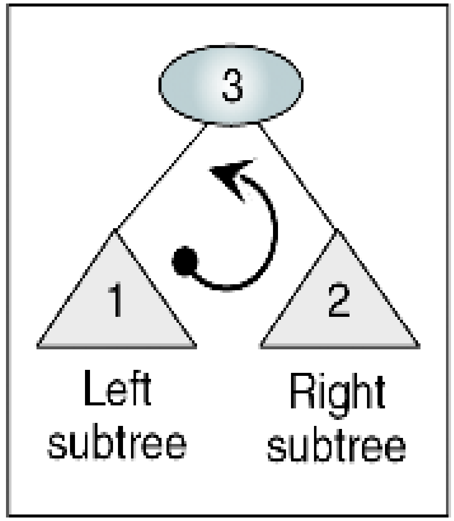
- Used to display/access the data in a tree in a certain order.
- In traversing always right sub-tree is traversed after left sub-tree.
- Three methods of traversing
    - **Preorder Traversing**
        - Root – Left –Right
    - **Inorder Traversing**
        - Left – Root – Right
    - **Postorder Traversing**
        - Left – Right - Root



(a) Preorder traversal



(b) Inorder traversal



(c) Postorder traversal

Preorder Traversal

- **Preorder traversal**
  - **Node – Left – Right**
  - **Prefix expression**
    - **++a*bc*+*defg**

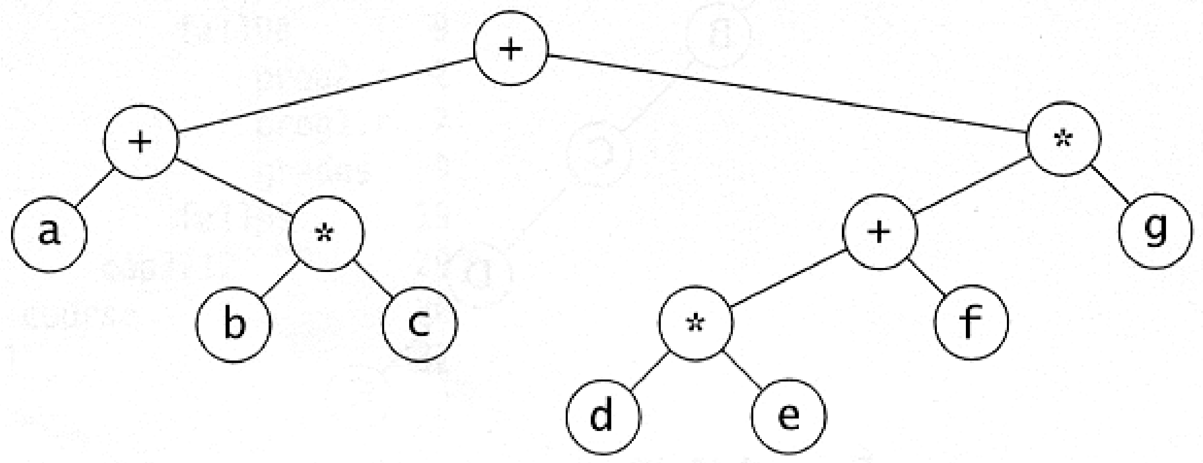


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Inorder Traversal

- **Inorder traversal**
  - **left, node, right.**
  - **infix expression**
    - **a+b*c+d*e+f*g**

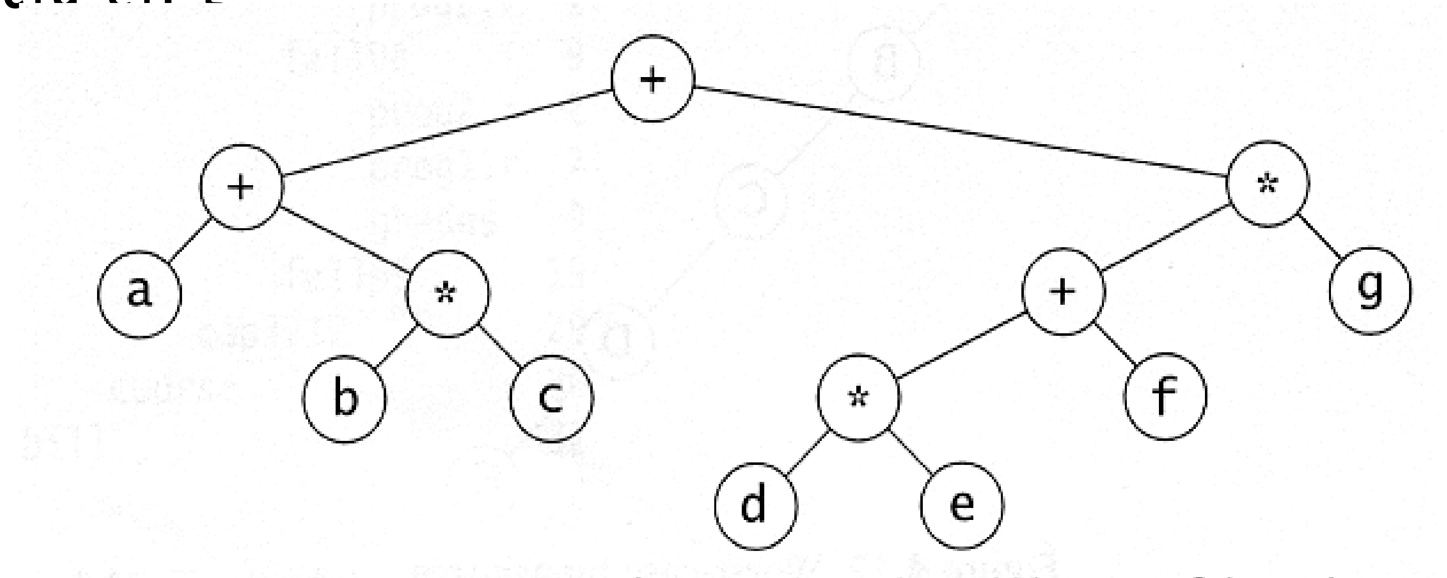


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Postorder Traversal

- **Postorder Traversal**
  - **left, right, node**
  - **postfix expression**
    - **abc*+d**

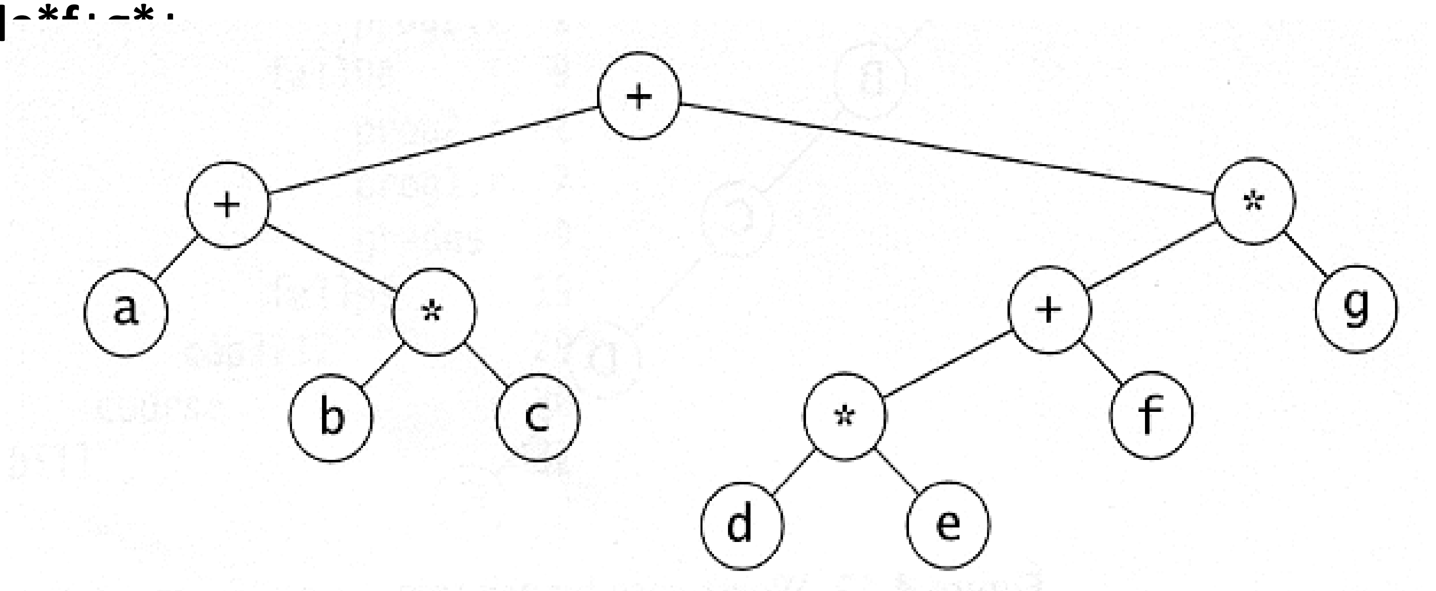
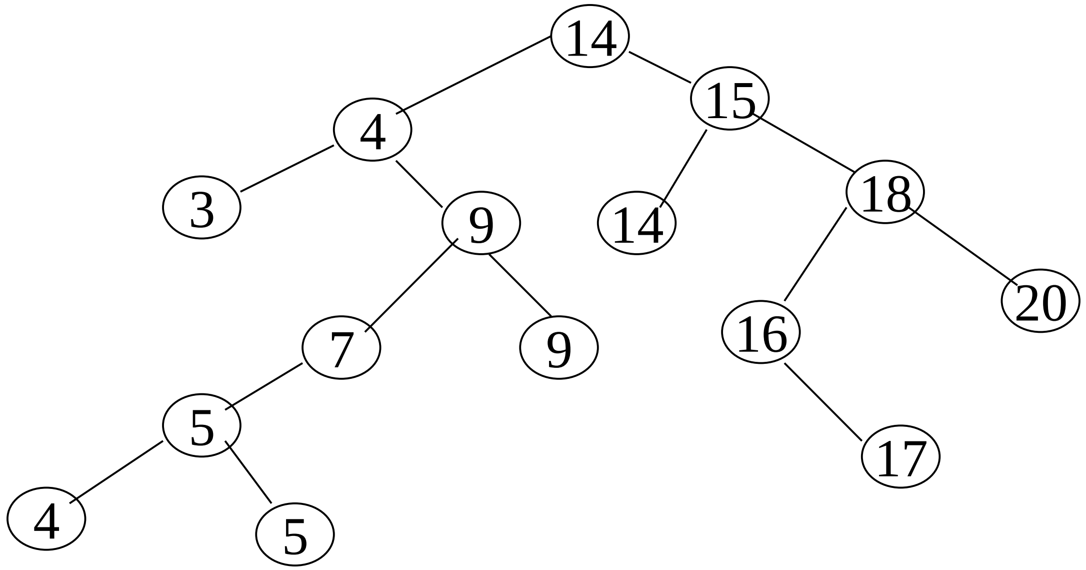


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Traversal Exercise

Traverse the following tree.



Pre-order Traversal?

Post-order Traversal?

In-order Traversal?

Inorder: 3, 4, 4, 5, 5, 7, 9, 9, 14, 14, 15, 16, 17, 18, 20

Post-order Traversal?

In-order Traversal?

Inorder Traversal Function

```
Void InorderTraversal(node* temp)
{
   if(temp!=NULL)
  {
            InorderTraversal(temp->LTree);
            cout<< temp->data;
            InorderTraversal(temp->RTree);
  }
}
```

Postorder Traversal Function

```
Void PostorderTraversal(node* temp)
{
   if(temp!=NULL)
   {
           PostorderTraversal(temp->LTree);
           PostorderTraversal(temp->RTree);
           cout<< temp->data;
   }
}
```

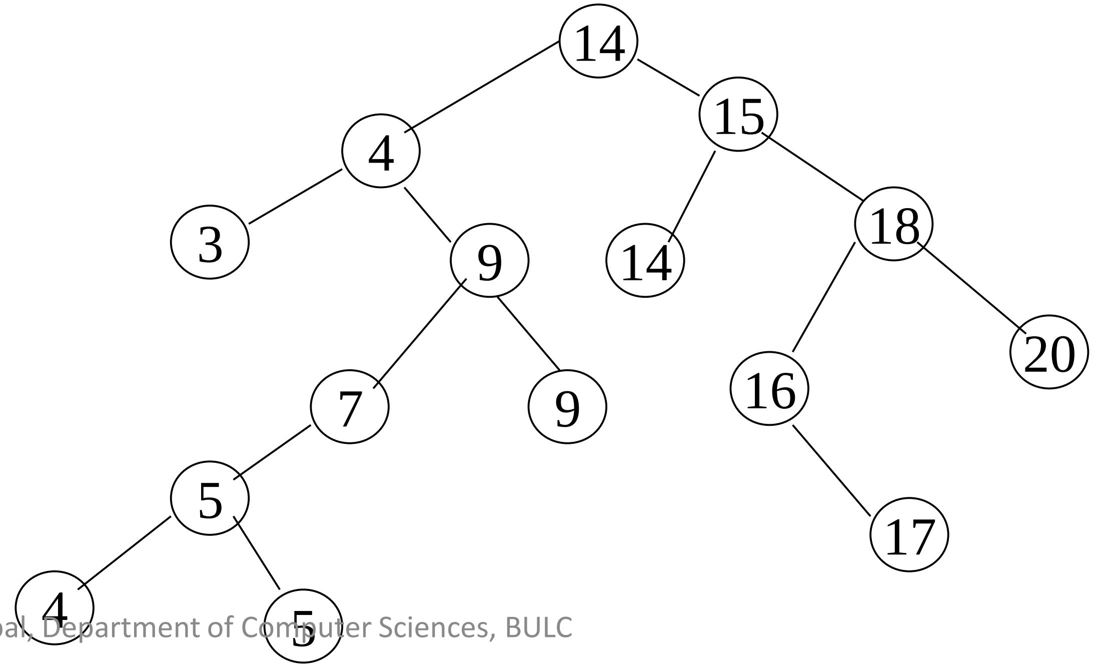
Preorder Traversal Function

```
Void PreorderTraversal(node* temp)
{
    if(temp!=NULL)
   {
            cout<<temp->data;
            PreorderTraversal(temp->LTree);
            PreorderTraversal(temp->RTree);
   }
}
```

Find the output …

14 4 3 3 9 7 5 4 4 5 5 5 7 9 9 9 4 15 14 14 18 16 17 17 16 20 20 18 15 14

```
void Traversal(node* temp)
{
   if(temp!=NULL)
  {
     cout<<temp->data;
     Traversal(temp->LTree);
      Traversal(temp->RTree);
       cout<<temp->data;
  }
}
```

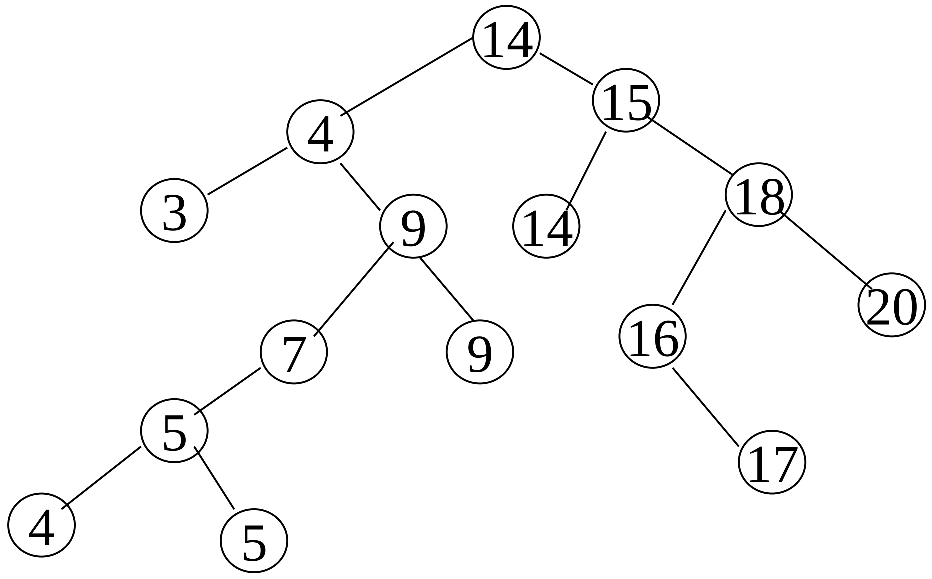


Find the output …

3 9 5 14 18 17

```
void Traversal(node* temp, int count)
{
   if(temp!=NULL)
  {
     if (count%2 == 0)
        cout<<temp->data;
     count++;
      Traversal(temp->Ltree, count);
      Traversal(temp->Rtree, count);


  }
}
```



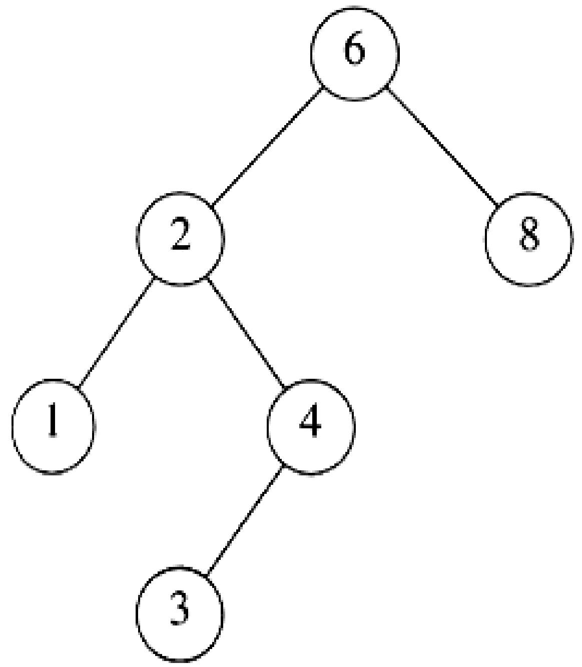
Binary Search Tree

Binary Search Tree

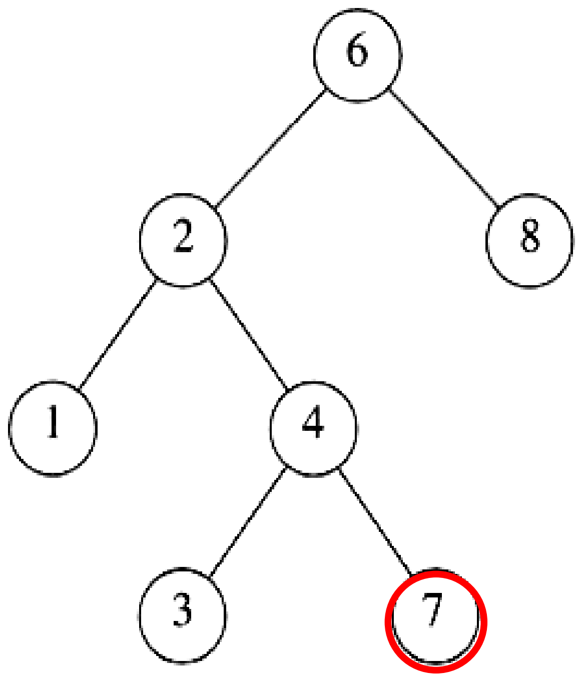
- **Stores keys in the nodes in a way so that searching, insertion and deletion can be done efficiently.**

- **Binary search tree is either empty or each node N of tree satisfies the following property**
  - **The Key value in the left child is not more than the value of root**
  - **The key value in the right child is more than or identical to the value of root**
  - **All the sub-trees, i.e. left and right sub-trees follow the two rules mention above.**

Examples



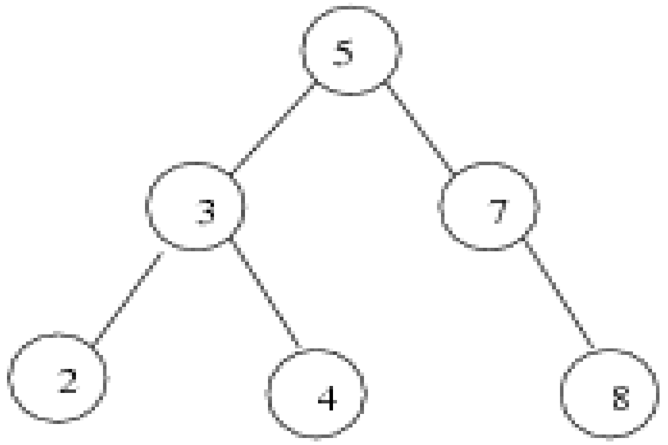
A binary search tree

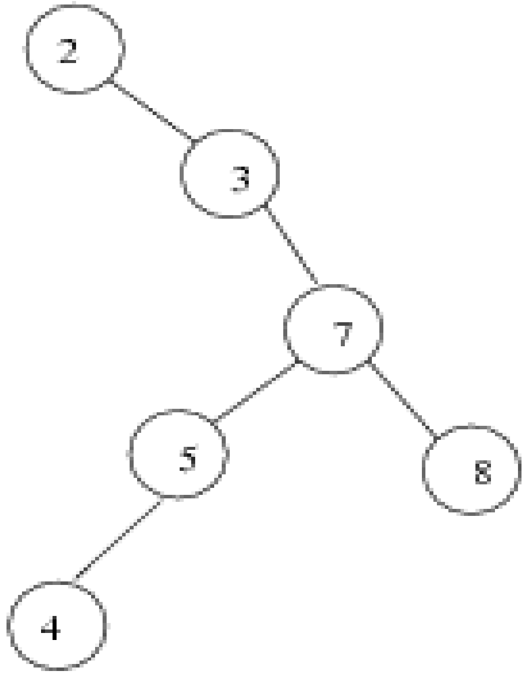


Not a binary search tree

Example 2

Two binary search trees representing the same Data set:

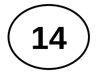




Example

- Input list of numbers:

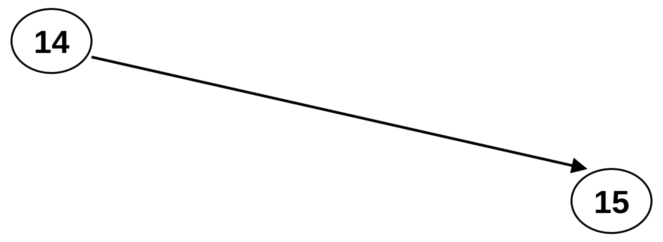
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

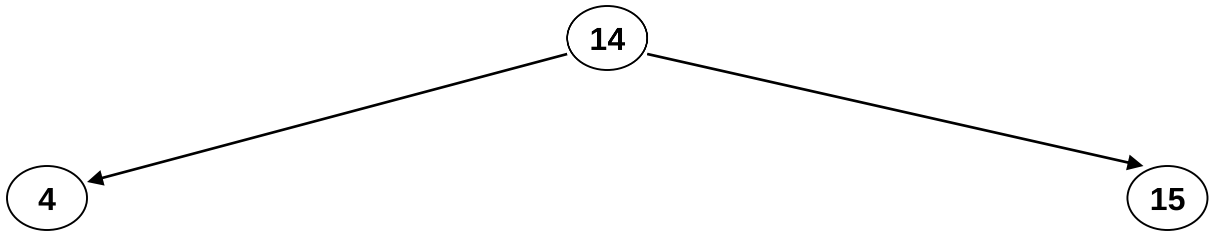
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

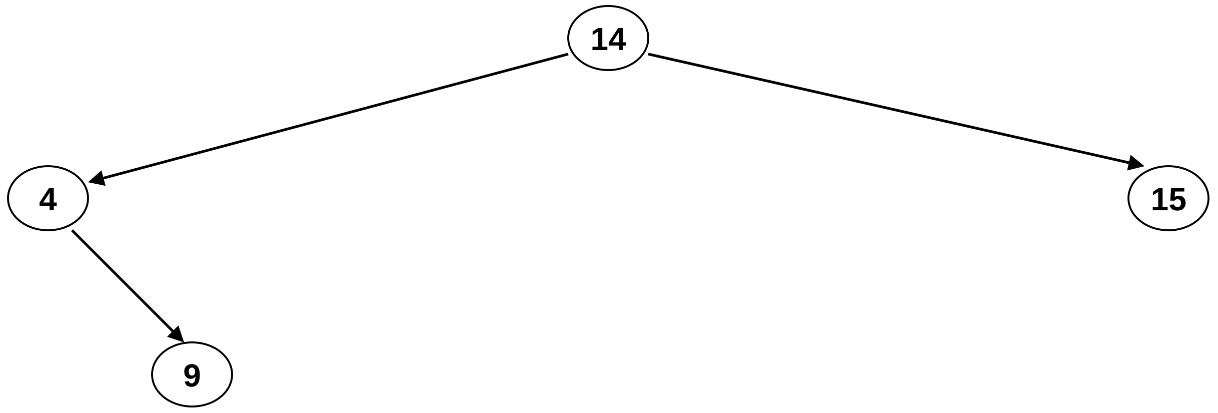
  14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

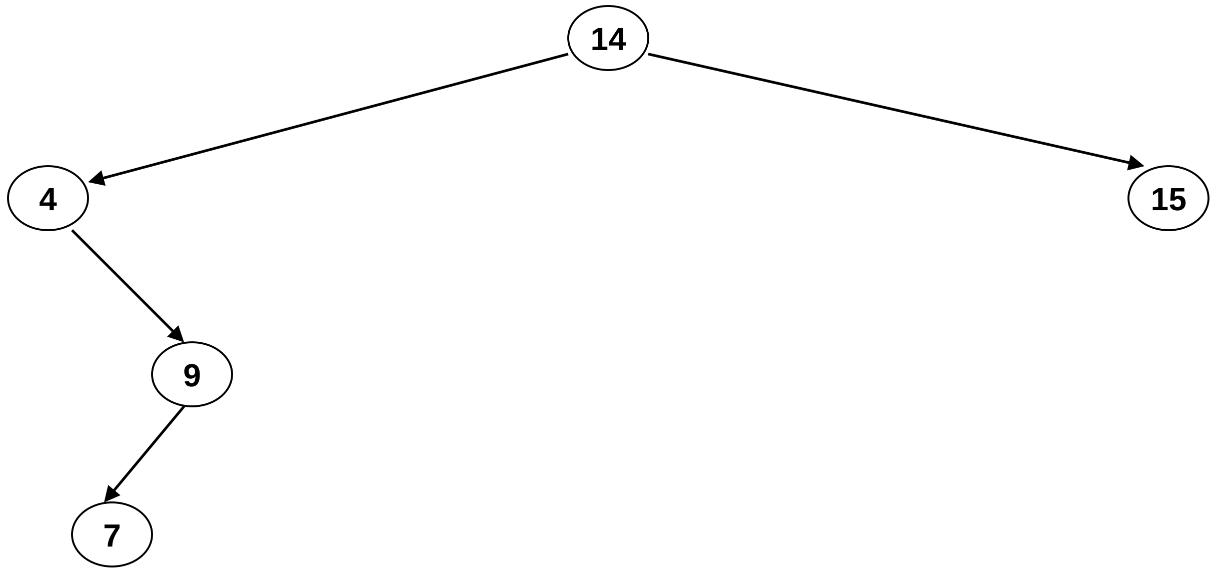
  14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

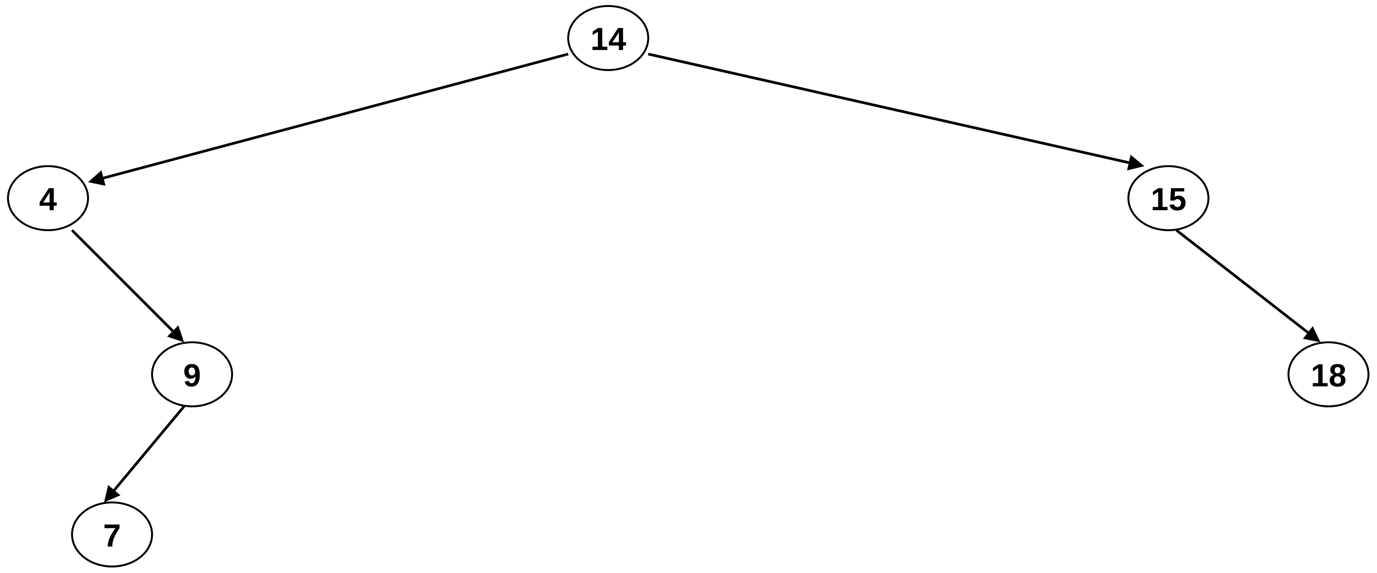
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

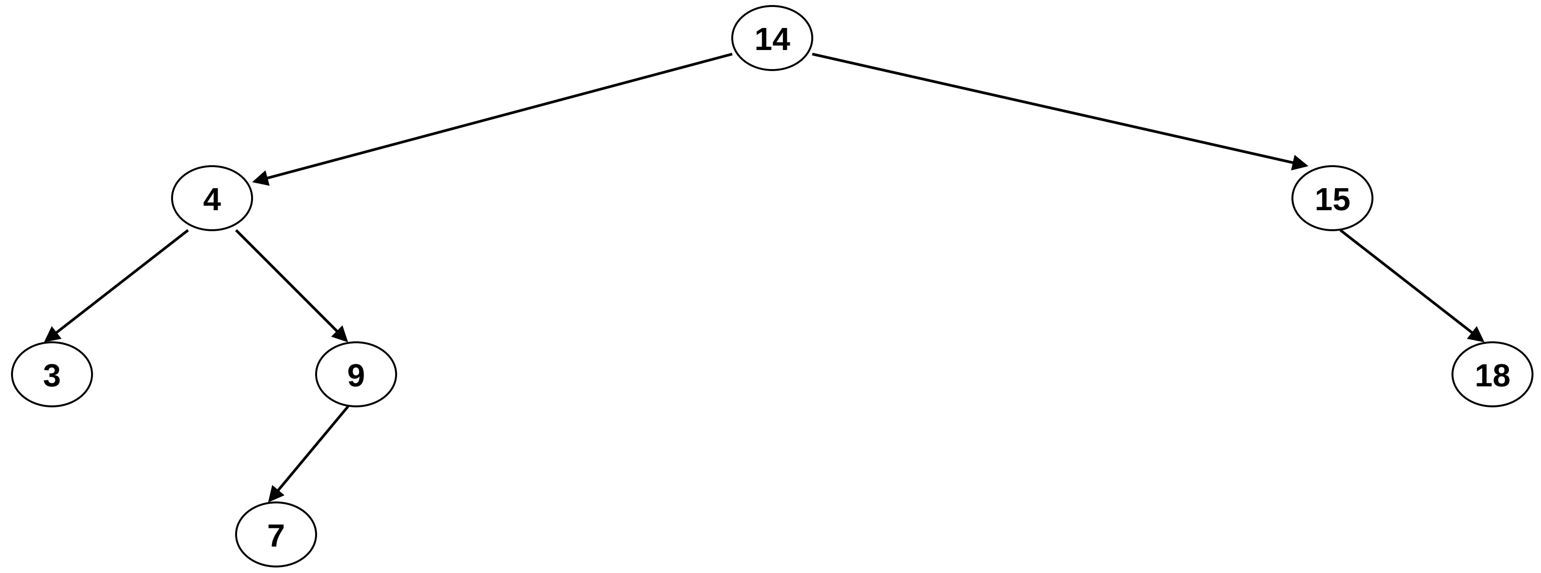
  14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

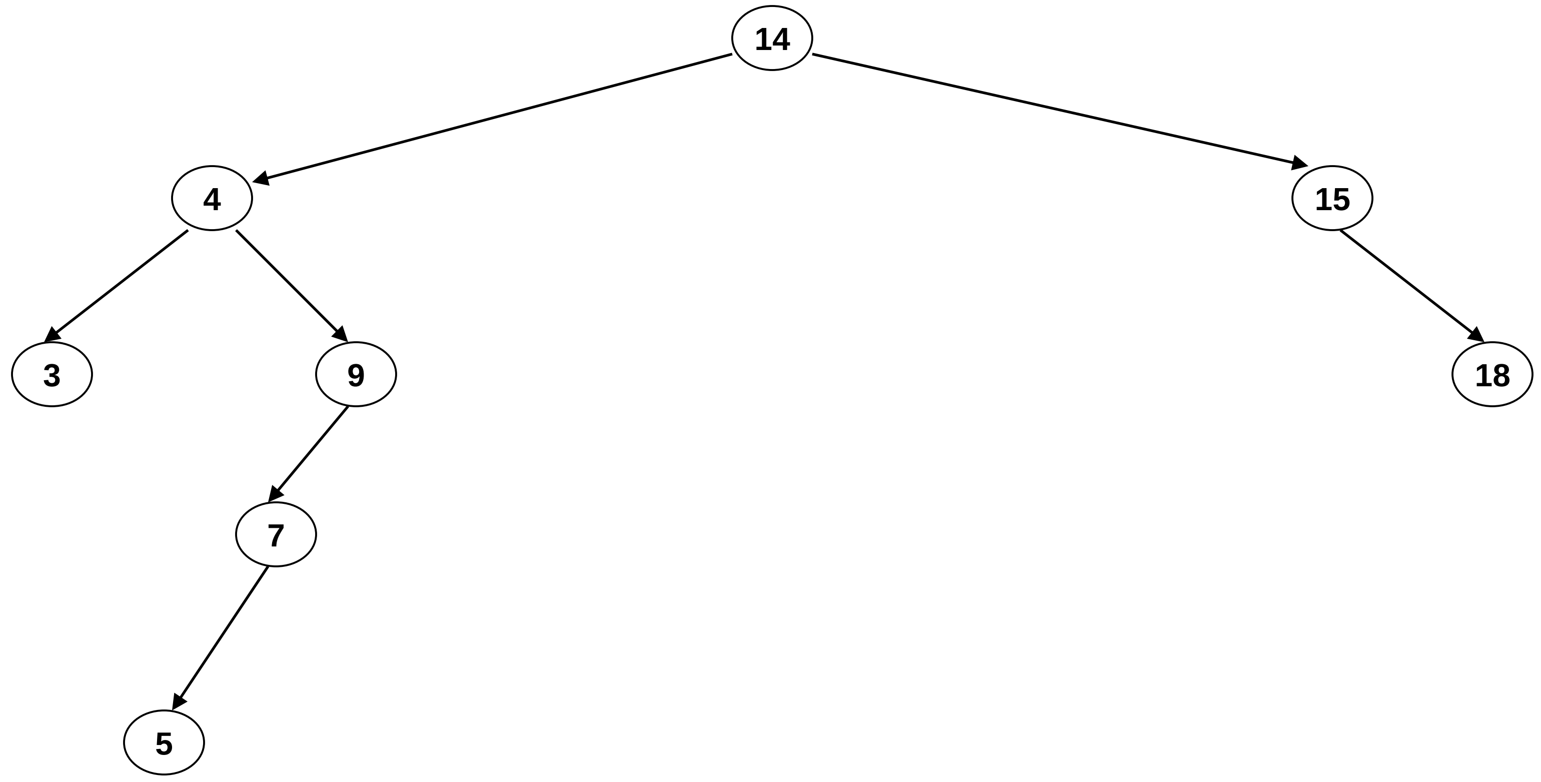
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

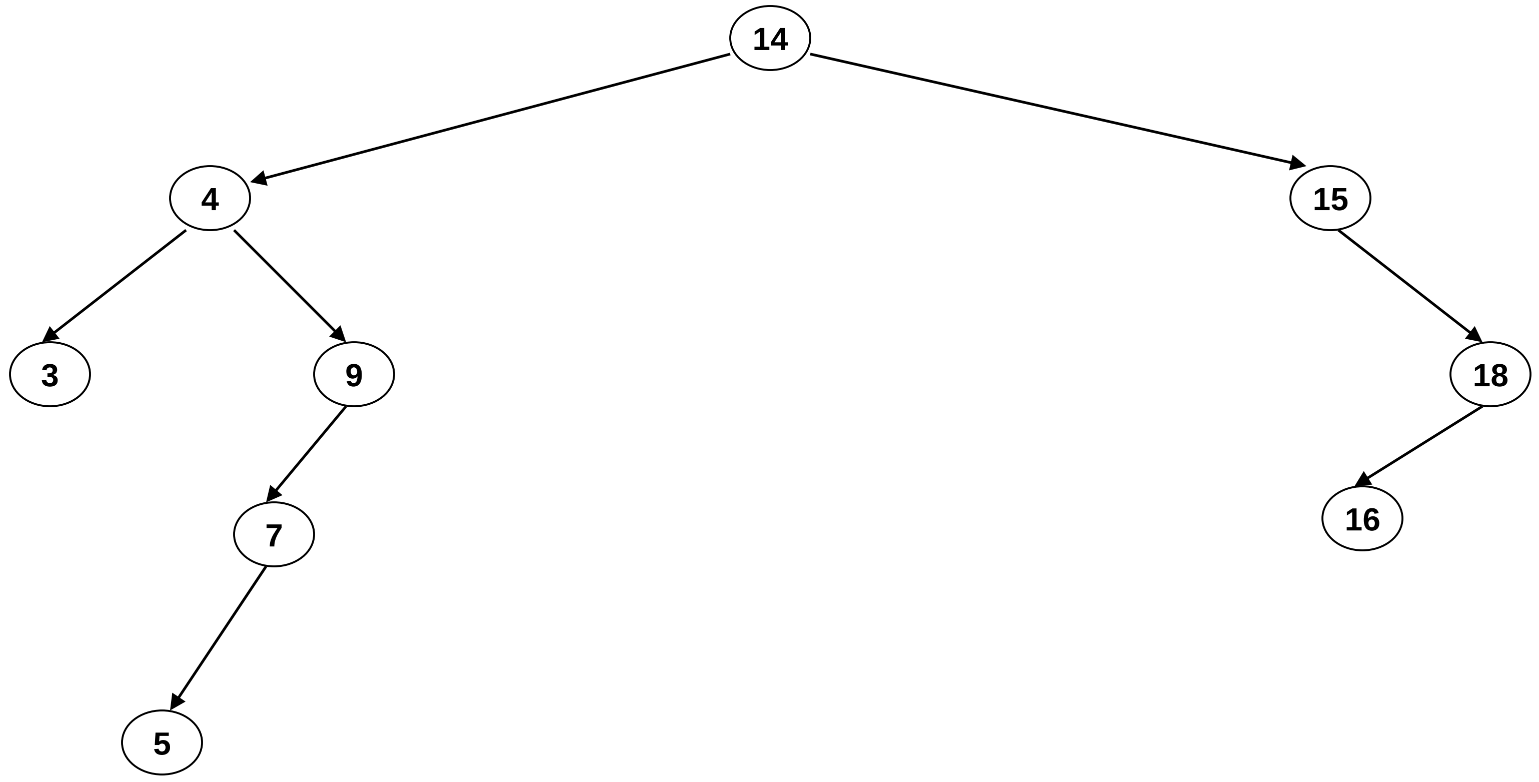
  14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

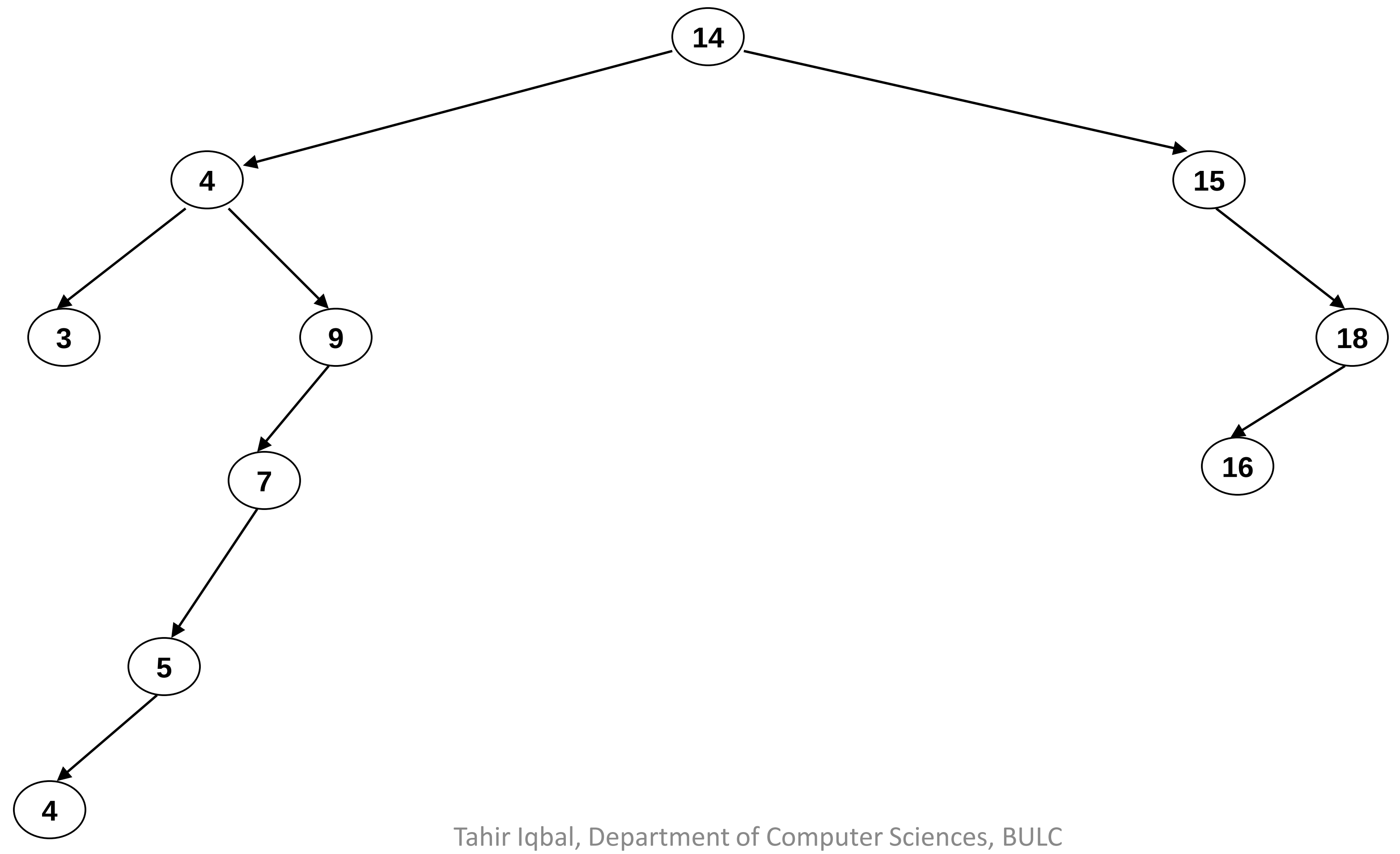
  14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

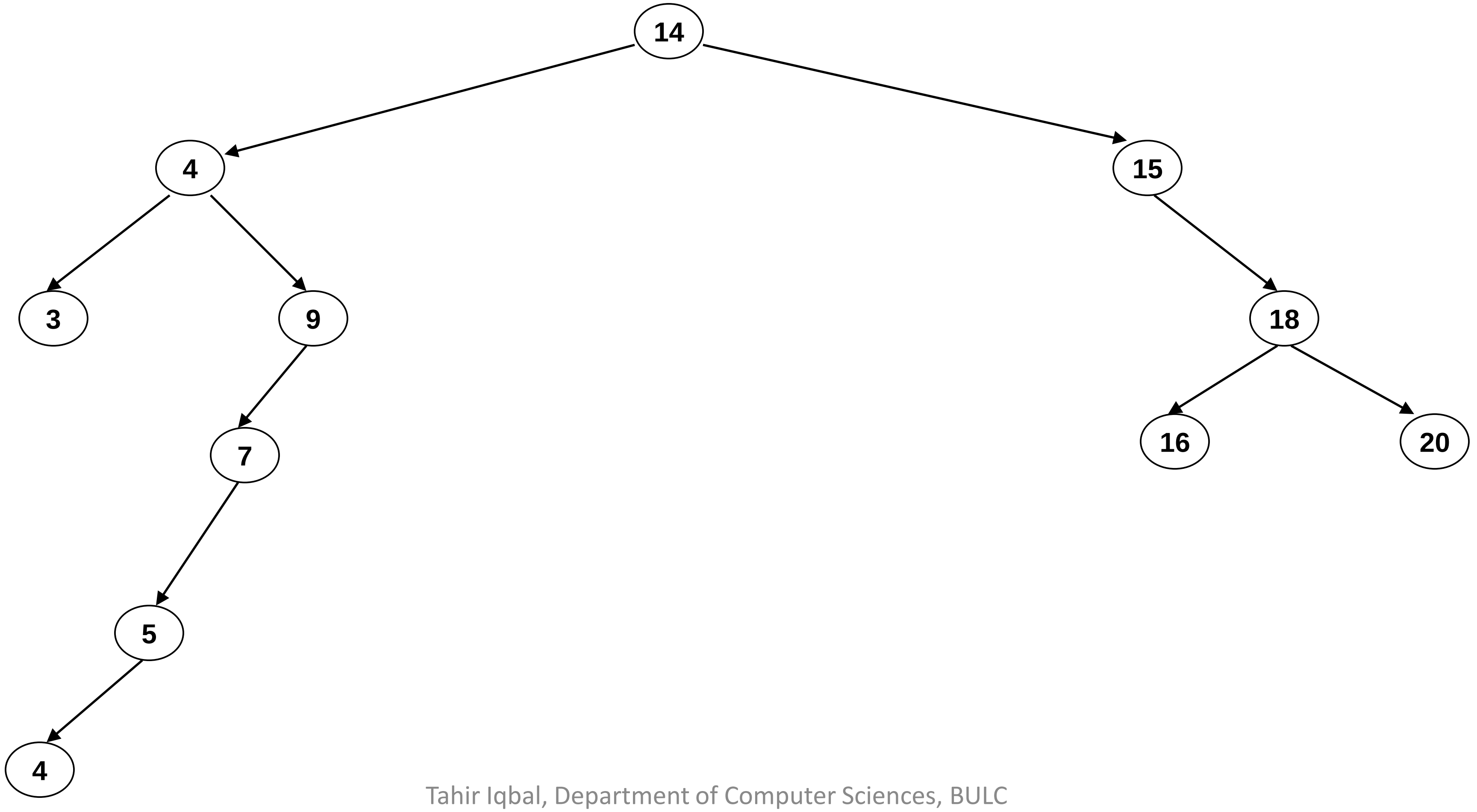
  14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

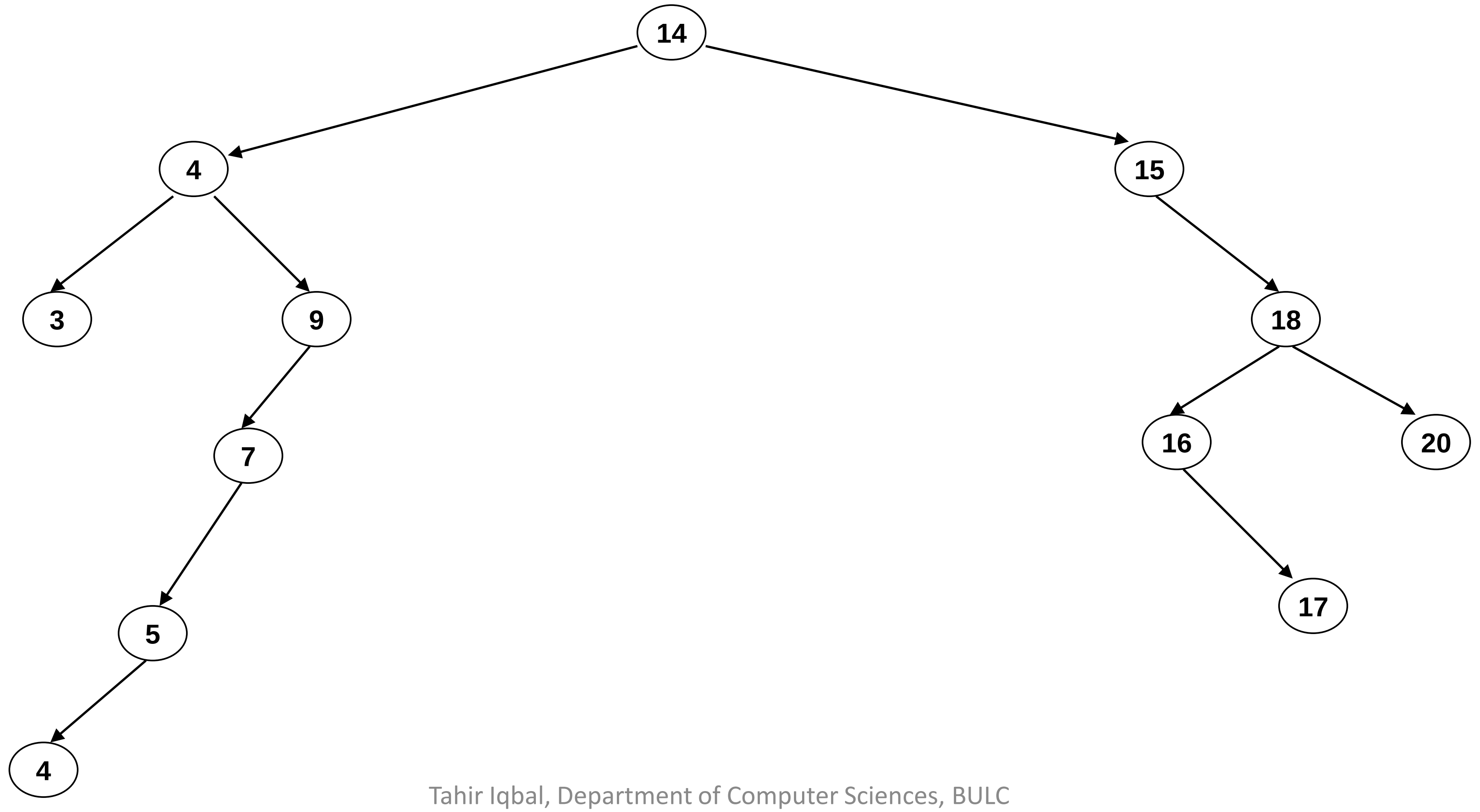
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

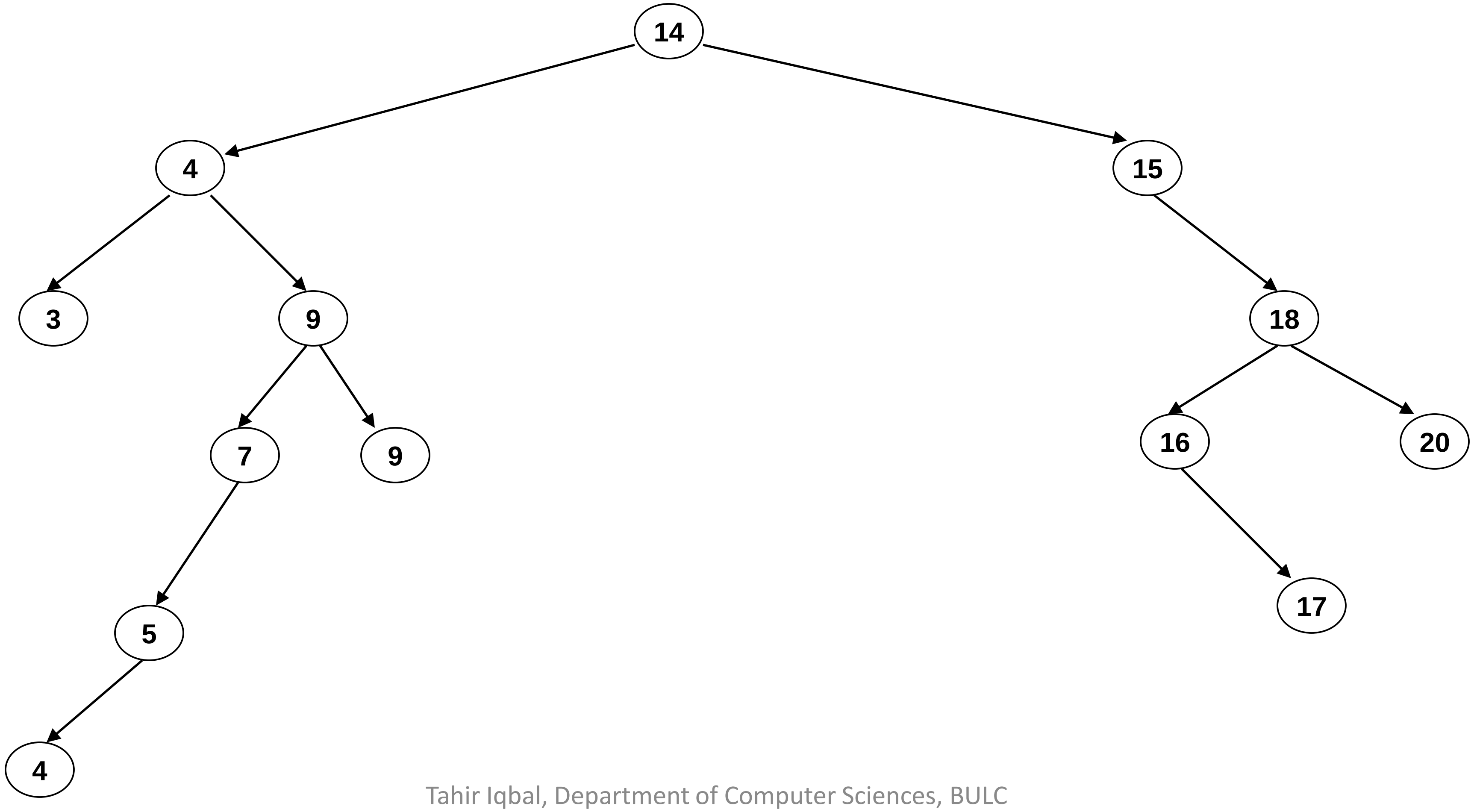
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

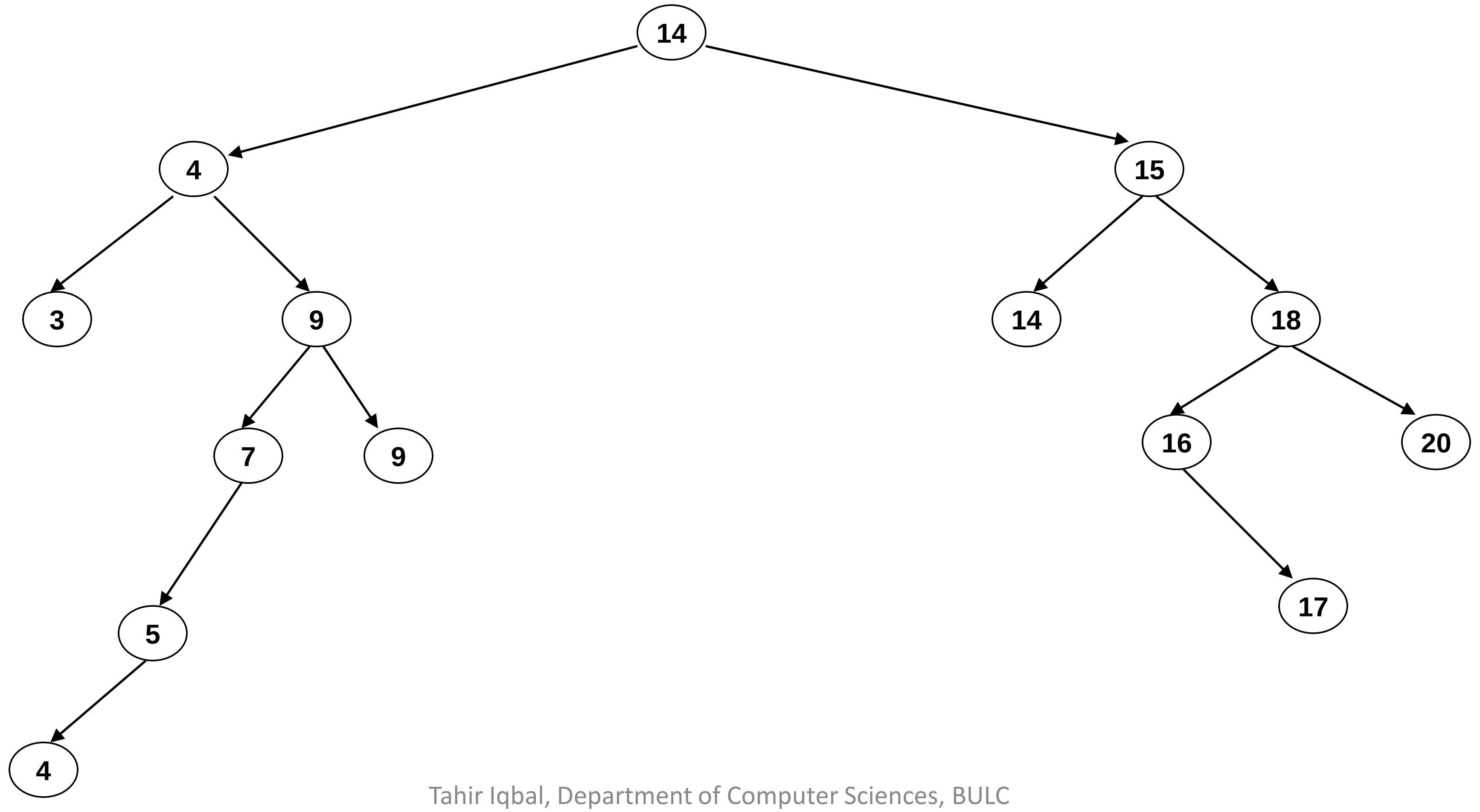
  14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

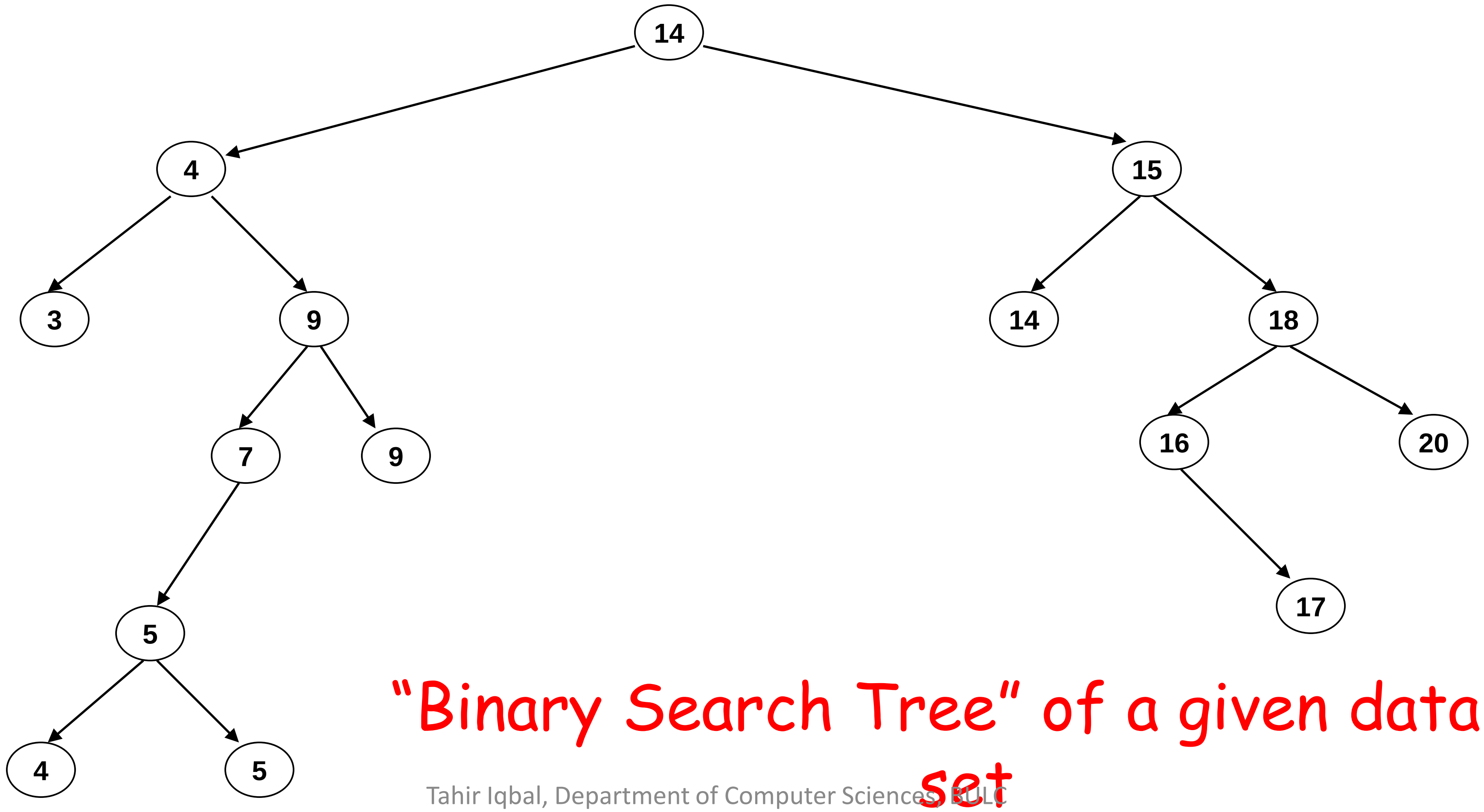
14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



Example

- Input list of numbers:

14 15 4 9 7 18 3 5 16 4 20 17 9 14 5



"Binary Search Tree" of a given data set

Binary Tree Implementation

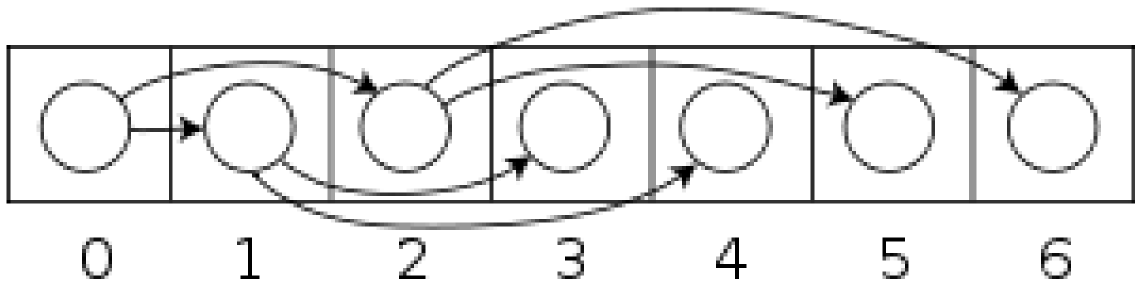
Binary Trees can be implemented using

arrays.

Left Child: 2i + 1

Right Child: 2i+2

This method of storage is often used for binary heaps.



Binary Tree Implementation

```
struct node{
                int data;
                node* LTree;
                node* RTree;
       };
node* root = NULL;
 node* Insert(node* , int);
 void Search(node* , int);
 void InorderTraversal(node*);
 void PreorderTraversal(node*);
 void PostorderTraversal(node*);
```

Insertion in BST

- **Three steps of insertion**
  - **If the root of the tree is NULL then insert the first node and root points to that node.**
  - **If the inserted number is lesser than the root node then insert the node in the left sub-tree.**
  - **If the inserted number is greater than the root node then insert the node in the right sub-tree.**

Insertion Function

```
node* Insert(node* temp, int num)
{
   if ( temp == NULL )
   {
            temp = new node;
            temp->data= num;
            temp->LTree= NULL;
            temp->RTree=NULL;
   }
   else if(num < temp->data)
            temp->LTree = Insert(temp->LTree, num);
   else //if( num >= temp->data)
            temp->RTree = Insert(temp->RTree,num);
   return temp;
}
```

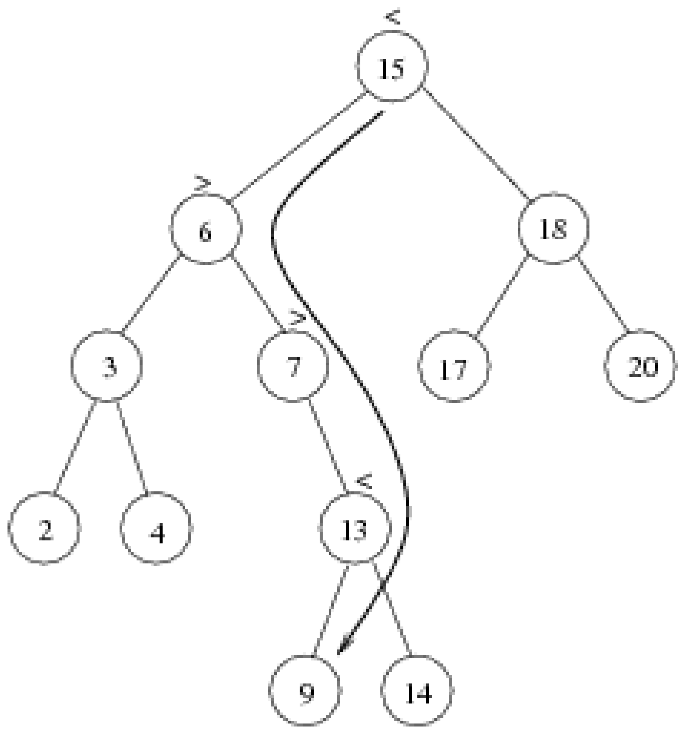
Searching in Binary Search Tree

- Three steps of searching
  - The item which is to be searched is compared with the root node. If the item is equal to the root, then we are done.
  - If its less than the root node then we search in the left sub-tree.
  - If its more than the root node then we search in the right sub-tree.
- The above process will continue till the item is found or you reached end of the tree.

Example: Search for 9 ...

Search for 9:

1. compare 9:15(the root), go to left subtree;
2. compare 9:6, go to right subtree;
3. compare 9:7, go to right subtree;
4. compare 9:13, go to left subtree;
5. compare 9:9, found it!



Search Function

```
Void Search(node* temp, int num)
{
   if(temp==NULL)
            cout<<“Number not found";
   else   if(temp->data == num)
                        cout<<"Number found";
   else if(temp->data > num)
                        Search(temp->LTree, num);
   else if(temp->data < num)
                        Search(temp->RTree, num);
}
```

Exercise

Write a function that finds a node from the tree and returns its parent’s address

```
node* getParent(node* curr, int num)
{
   if((curr->LTree->data == num) ||
(curr->RTree->data == num))
      return curr;
   else if(num < curr->data)
      return getParent(curr->LTree,
num);
   else //if(num > curr->data)
      return getParent(curr->RTree,
num);
}
```

```
node* getParent(node* curr, node* prev, int
num)
{
   if(curr->data == num)
      return prev;
   else if(num < curr->data)
     retun getParent(curr->LTree, curr, num);
   else //if(num > curr->data)
     return getParent(curr->RTree, curr,
num);
}
```

Exercise

Write a function that finds the node with minimum value.

```
node* FindMin(node* curr)
{
   if(curr->LTree == NULL)
      return curr;
   else
      return FindMin(curr->LTree);
}
```

Deletion in BST

- When we delete a node, we need to consider how we take care of the children of the deleted node.
    - This has to be done such that the property of the search tree is maintained.

Deletion in BST

Three cases:

(1) The node is a leaf

- Delete it immediately

(2) The node has one child

- Adjust a pointer from the parent to bypass that node

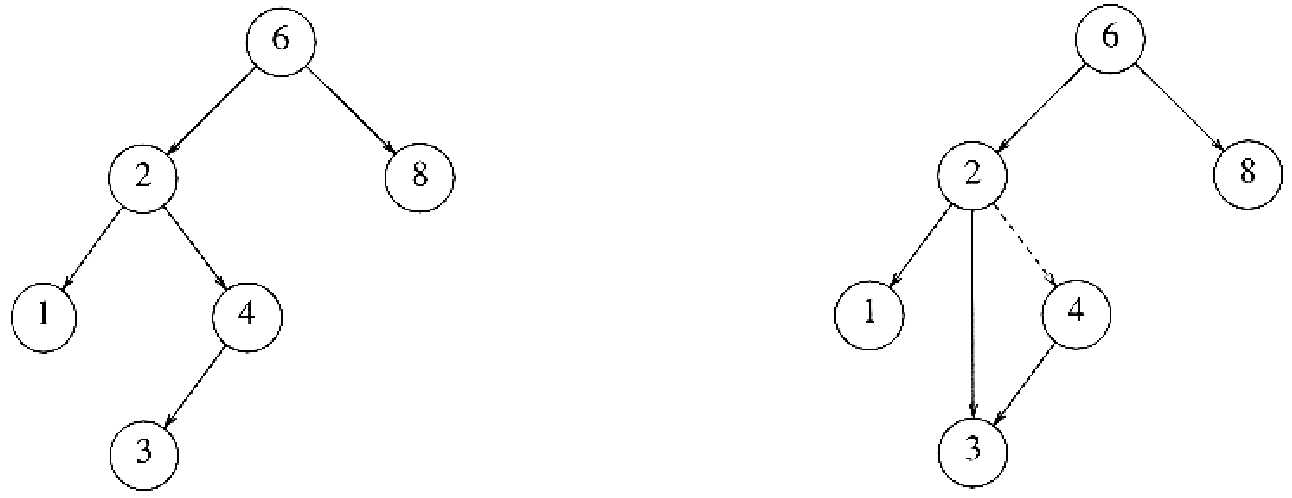


Figure 4.24 Deletion of a node (4) with one child, before and after

Deletion in BST

(3) The node has 2 children

- Replace the key of that node with the minimum element at the right subtree
- Delete the minimum element
    - Has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen. So invoke case 1 or 2.

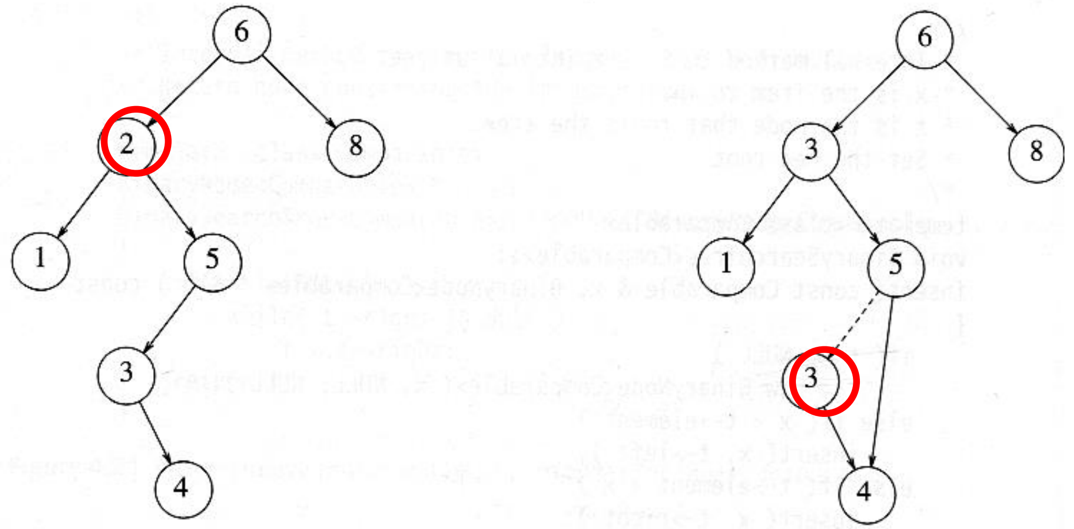


Figure 4.25 Deletion of a node (2) with two children, before and after

Deletion Code

```
Void DeleteNode(node* temp, int num)
{   if (temp==NULL)
    cout<<"Number not Found";
  else if((temp->data == num)) //node found
  {     node *parent, *min ;
        int number;
    // if number is found at a leaf node
    if((temp->LTree == NULL) && (temp->RTree == NULL))
    {
            parent=GetParent(root, temp->data, root);  //will return parent node
            if(parent->LTree == temp)
                    parent->LTree = NULL;
            else if (parent->RTree == temp)
                    parent->RTree = NULL;
          delete temp;
    }
```

```
// if node to be deleted has one child
    else if(((temp->LTree == NULL) && (temp->RTree != NULL)) || ((temp->LTree != NULL)
    && (temp->RTree == NULL)))
    {
            parent = GetParent(root, temp->data, root); //will return parent node
            if(temp->LTree != NULL){
                    if(parent->LTree == temp)
                            parent->LTree = temp->LTree;
                    else if (parent->RTree == temp)
                            parent->RTree = temp->LTree;
             }
            else if(temp->RTree != NULL){
                    if(parent->LTree == temp)
                            parent->LTree = temp->RTree;
                    else if (parent->RTree == temp)
                            parent->RTree = temp->RTree;
            }
            delete temp;
    }
```

```
//if node to be deleted has two children
else if((temp->LTree != NULL) && (temp->RTree != NULL))
    {
      min = FindMin(temp->RTree);  //will return the min. no. found in RTree
     number = min->data;
      DeleteNode(temp->RTree, min->data);  //calling to itself recursively
      temp->data= number;
    }
  } // end node found



  else if (num < temp->data)
   DeleteNode(temp->LTree, num);   //calling to itself recursively
  else  //if (num > temp->data)
   DeleteNode(temp->RTree, num); //calling to itself recursively
}
```

Animations

- http://www.cs.jhu.edu/~goodrich/dsa/trees/btree.html
- http://www.cosc.canterbury.ac.nz/mukundan/dsal/appldsal.html

Parse Tree

The first step in building a parse tree is to break up the expression string into a list of tokens. There are four different kinds of tokens to consider: left parentheses, right parentheses, operators, and operands. We know that whenever we read a left parenthesis we are starting a new expression, and hence we should create a new tree to correspond to that expression. Conversely, whenever we read a right parenthesis, we have finished an expression. We also know that operands are going to be leaf nodes and children of their operators. Finally, we know that every operator is going to have both a left and a right child.

Using the information from above we can define four rules as follows:

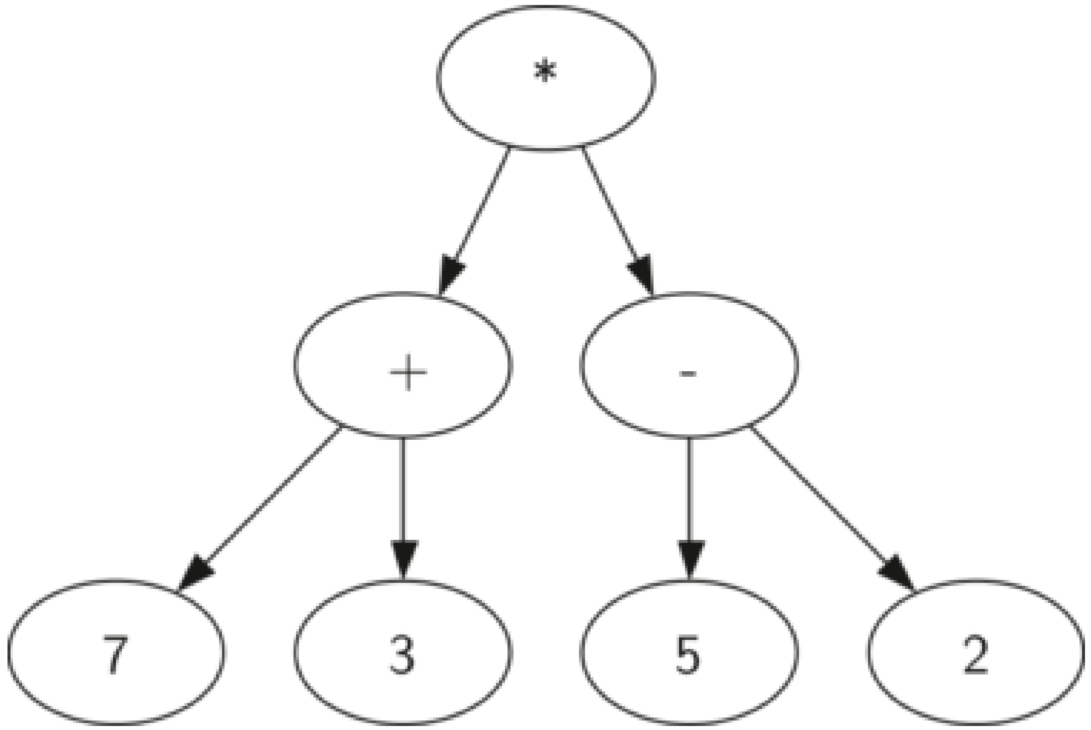
If the current token is a '(', add a new node as the left child of the current node, and descend to the left child.

If the current token is in the list ['+','-','/','*'], set the root value of the current node to the operator represented by the current token. Add a new node as the right child of the current node and descend to the right child.

If the current token is a number, set the root value of the current node to the number and return to the parent.

If the current token is a ')', go to the parent of the current node.

((7+3)*(5-2))



Iterative method for Inorder Traversal

```
void
in_order_traversal_iterative(Binar
yTree *root) {

struct stack s;
BinaryTree *current = root;
bool done = false;
while (!done) {
    if (current) {
        s.push(current);
        current = current->left;
    }
    else {
        if (s.empty()) {
            done = true;
        } else {
            current = s.top();
            s.pop();
            cout << current->data << " ";
            current = current->right;
        }
    }
  } // endwhile
}
```

Iterative method for Inorder Traversal

```
void in_order_traversal_iterative(BinaryTree *root) {
    stack<BinaryTree*> s;
    BinaryTree *current = root;
    while (!s.empty() || current) {
      if (current) {
          s.push(current);
          current = current->left;
      } else {
          current = s.top();
          s.pop();
          cout << current->data << " ";
          current = current->right;
      }
    } // endwhile
}
```