Chapter 07
Trees

By Kibret Zewdu

Objectives

- After completing this chapter, you will be able to understand the following:
  - Hierarchical representation of data using trees
  - Binary search trees (BSTs) that allow both rapid retrievals by key and inorder traversals
  - The use of trees as a flexible data structure for solv ing a wide range of problems

Introduction

- In computer science, a *tree* is a widely used data structure that emulates a tree structure with a set of linked nodes.
- Trees are used popularly in computer programming.
- They can be used for improving database search times, in game programming, 3D graphics programming, arithmetic scripting languages (arithmetic precedence trees), data compression (Huffman trees), and even file systems.

Introduction

- A data structure is said to belinear if its elements form a sequence or a linear list.
  - Every data element has a unique successor and a unique predecessor.
  - There are two basic ways of representing linear structures in memory ardinked lists and sequential organization, that is, arrays .
- Non-linear data structures are used to represent the data containing hierarchical or network relationship between the elements.
  - Trees and graphs are examples of non-linear data structures.
  - Every data element may have more than one predecessor as well as successor.
  - Elements do not form any particular linear sequence.

Introduction…

- Wherever the hierarchical relationship among data is to be preserved, the **tree** is used.
- Well-known examples of such structures are *family trees* , *hierarchy of positions in organization* , and so on.
- The *operating system* organizes folders and files using a tree structure

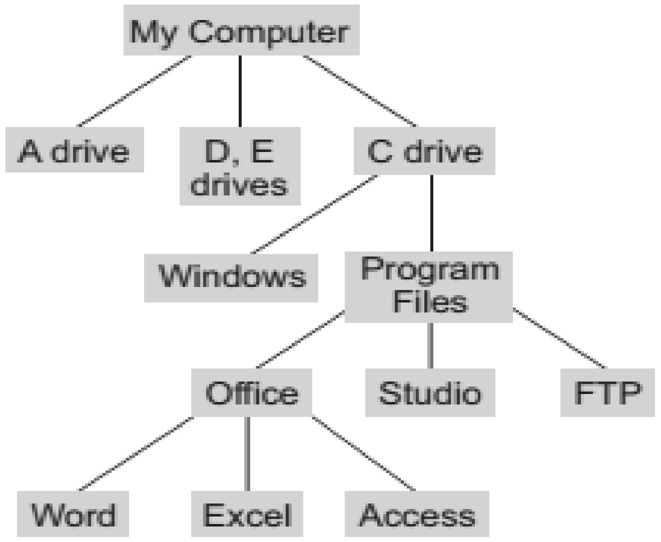


Fig. 7.1 Folder and subfolders organization

Introduction…

- The common uses of trees include the following:
  1. Manipulating hierarchical data
  2. Making information easily searchable
  3. Manipulating sorted lists of data

Basic Terminology

- ***Root :*** A directed tree has one node called its *root* , with indegree zero, whereas for all other nodes, the indegree is 1 .
- ***Terminal node (leaf node) :*** In a directed tree, any node that has an outdegree zero is a terminal node.
  - The terminal node is also called as leaf node (or external node) .
- ***Branch node (internal node) :*** All other nodes whose outdegree is not zero are called as branch nodes.
- ***Level of node:*** The level of any node is its *path length from the root.*
  - The level of the root of a directed tree is zero, whereas the level of any node is equal to its distance from the root.
  - Distance from the root is the number of edges to be traversed to reach the root.
- The **degree of a node** is the number of subtrees it has.
- The ***degree of a tree*** is the maximum degree of a node in the tree.

General Tree

- A tree T is defied recursively as follows:
  1. A set of zero items is a tree, called the ***empty tree*** (or NULL tree).
  2. If $T_1, T_2, \ldots, T_n$ are n trees for $n > 0$ and R is a node, then the set containing R and the trees $T_1, T_2, \ldots, T_n$ are a tree.
     - Within T, R is called the *root* of T, and $T_1, T_2, \ldots, T_n$ are called *subtrees*.

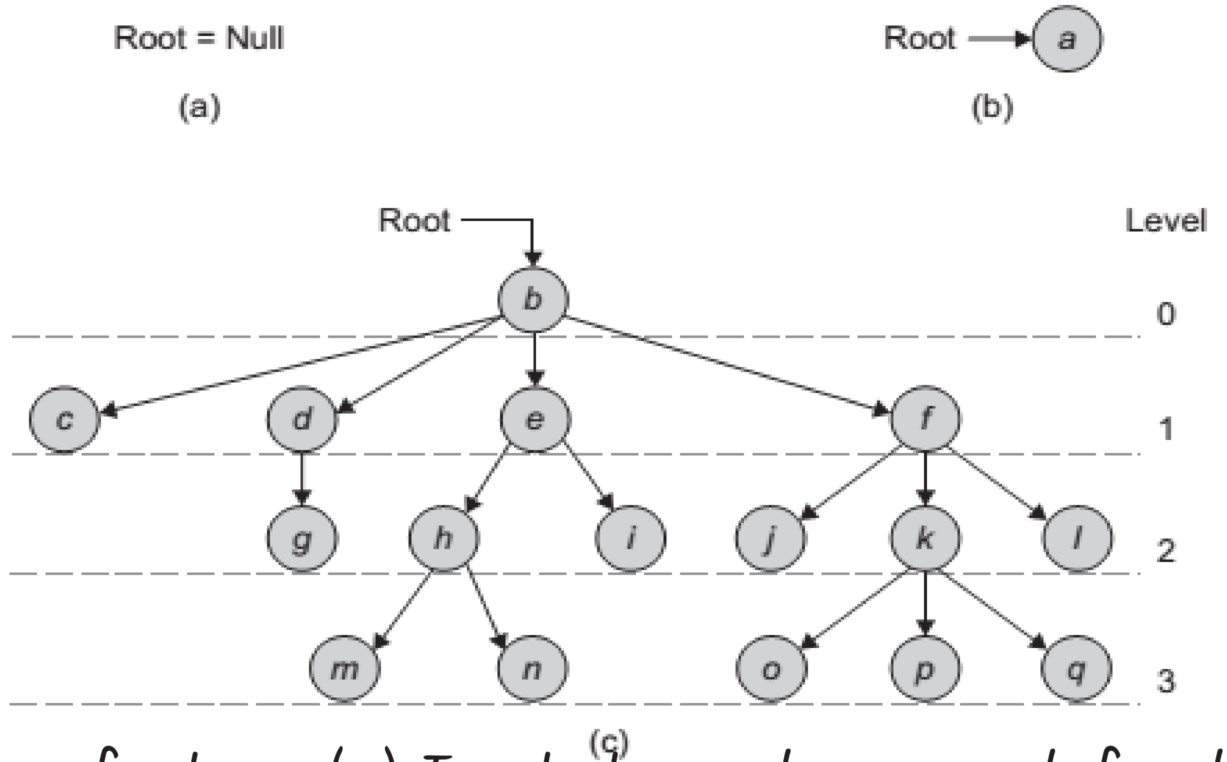


Fig. 7.8 Degree of a tree (a) Empty tree—degree undefined (b) Tree with a single node—degree 0 (c) Tree of height 3—degree 4

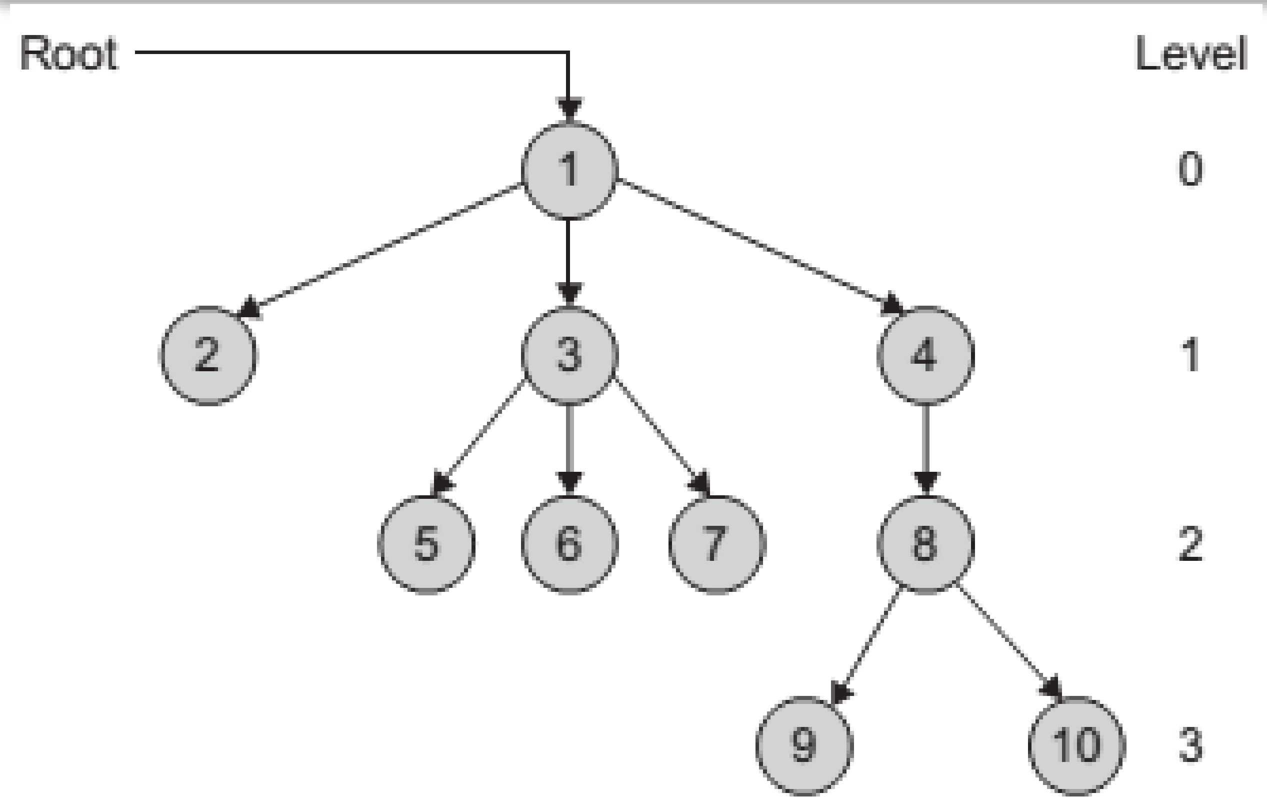
General Tree

- ***Family relationships*** can be modelled as trees, we often call the root of a tree (or subtree) the *parent* , and the roots of the subtrees the *children* .
- The children of the same node are called ***siblings*** .
- ***Paths :*** Paths exist from all parents to children. A unique path exists from the root to each leaf node.
- The ***length of a path*** is the number of edges it contains (which is one less than the number of nodes on the path).
- The ***height of a tree*** is the *length of the path from the root* to a node at the *lowest level* .
  - i.e, the height of a tree is the maximum path length in the tree.

Tree Traversal

- There are three common ways to symmetrically order (or list) the nodes in a tree: **preorder**, **inorder**, and **postorder**.
  1. The **preorder** list contains the root followed by the preorder list of nodes of the subtrees of the root from left to right.
  2. The **inorder** list contains the inorder list of the leftmost subtree, the root, and the inorderlist of each of the other subtrees from left to right.
  3. The **postorder** list contains the postorder list of subtrees of the root from left to right followed by the root.

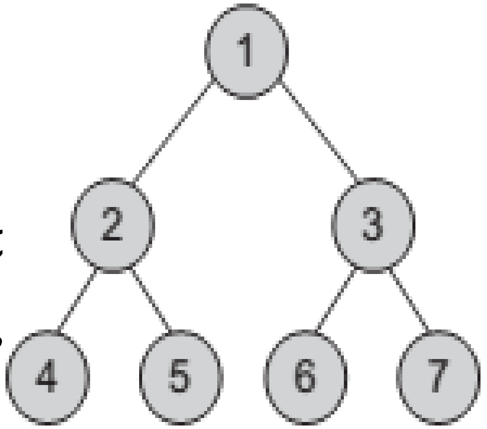
Tree Traversal



1 –> 2 –> 3 –> 5 –> 6 –> 7 –> 4 –> 8 –> 9 –> 10 (preorder)
2 –> 1 –> 5 –> 3 –> 6 –> 7 –> 9 –> 8 –> 10 –> 4 (inorder)
2 –> 5 –> 6 –> 7 –> 3 –> 9 –> 10 –> 8 –> 4 –> 1 (postorder)

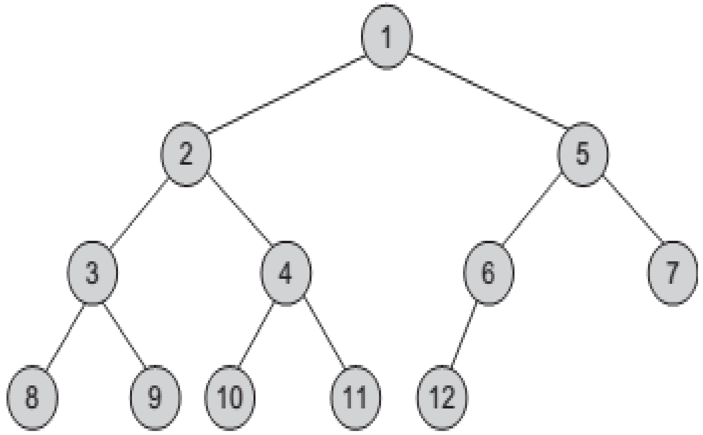
Types of Trees

- **Rooted tree:** a rooted tree where one node is designated as root, whose incoming degree is zero, whereas for all other nodes, the incoming degree is one
- **Binary tree:** no node has more than two children.
- **Full binary tree:** A binary tree is a full binary tree if it contains the maximum possible number of nodes in all levels.
  - Each node has two children or no child at all.
  - The total number of nodes in a full binary t h is 2h+1 – 1 considering the root at level 0.



Types of Trees

- ***Complete binary tree :*** A binary tree is said to be a complete binary tree if all its levels except the last level have the maximum number of possible nodes, and all the nodes of the last level appear as f***deft*** as possible.
  - All the leaf nodes are at the last and the second last level, and the levels are filed from left to right.
  - Applications such as the ***priority queue*** and ***heap sort*** use the complete binary tree.



Types of Trees

- ***Left skewed binary tree:*** If the right subtree is missing in every node of a tree, we call it *a*left skewed tree.
- **Right skewed tree:** If the left subtree is missing in every node of a tree, we call it asright subtree.

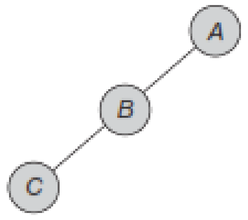


Fig: Left skewed tree

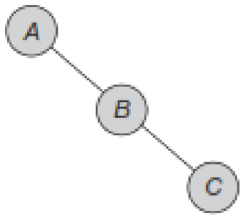
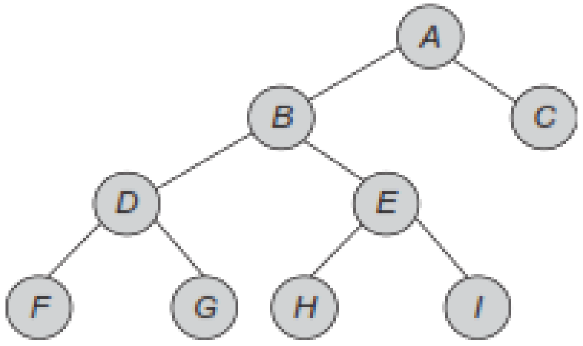


Fig: Right skewed tree

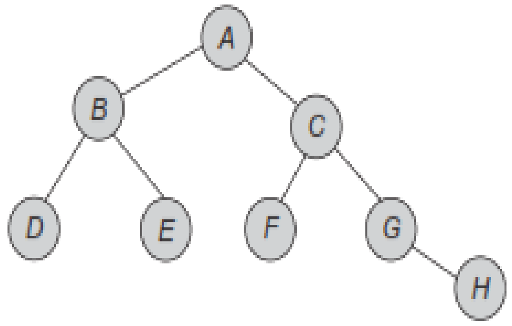
- ***Strictly binary tree:*** If every non-terminal node in a binary tree consists of non-empty left and right subtrees, then such a tree is called strictly binar

Fig : Strictly binary tre



Binary Tree

- One of the most commonly used classes of trees is a binary tree.
- A binary tree has the degree two, with each node having at most two children.
  - This makes the implementation of trees easier.
  - In addition, binary trees have a wide range of applications.
- **Definition:** A binary tree
  1. is either an empty tree or
  2. consists of a nod ıd two children, left and right, each o ı binary tree.



Operations on binary tree

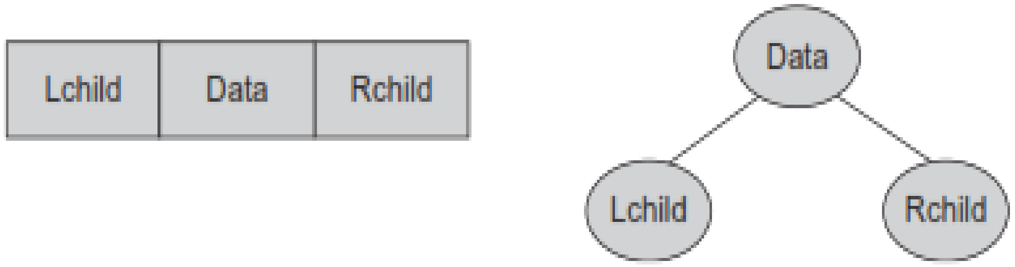
- The basic operations on a binary tree can be as listed as follows:
  1. Creation—Creating an empty binary tree to which the 'root' points
  2. Traversal—Visiting all the nodes in a binary tree
  3. Deletion—Deleting a node from a non-empty binary tree
  4. Insertion—Inserting a node into an existing (may be empty) binary tree
  5. Merge—Merging two binary trees
  6. Copy—Copying a binary tree

Implementation Of A Binary Tree

- The implementation of a binary tree should represent the hierarchical relationship between a parent node and its left and right children.
- The implementation can be both in array and linked list.

Linked Implementation Of Binary Trees

- Binary tree has a natural implementation in a linked storage.
- In a linked organization, we wish that all the nodes should be allocated dynamically.
- Each node with data and link fields.
  - Each node of a binary tree has both a left and a right subtree.
  - Each node will have three fields—Lchild, Data, and Rchild.
- A node does n



parent node.

Fig. Tree node

Linked Representation

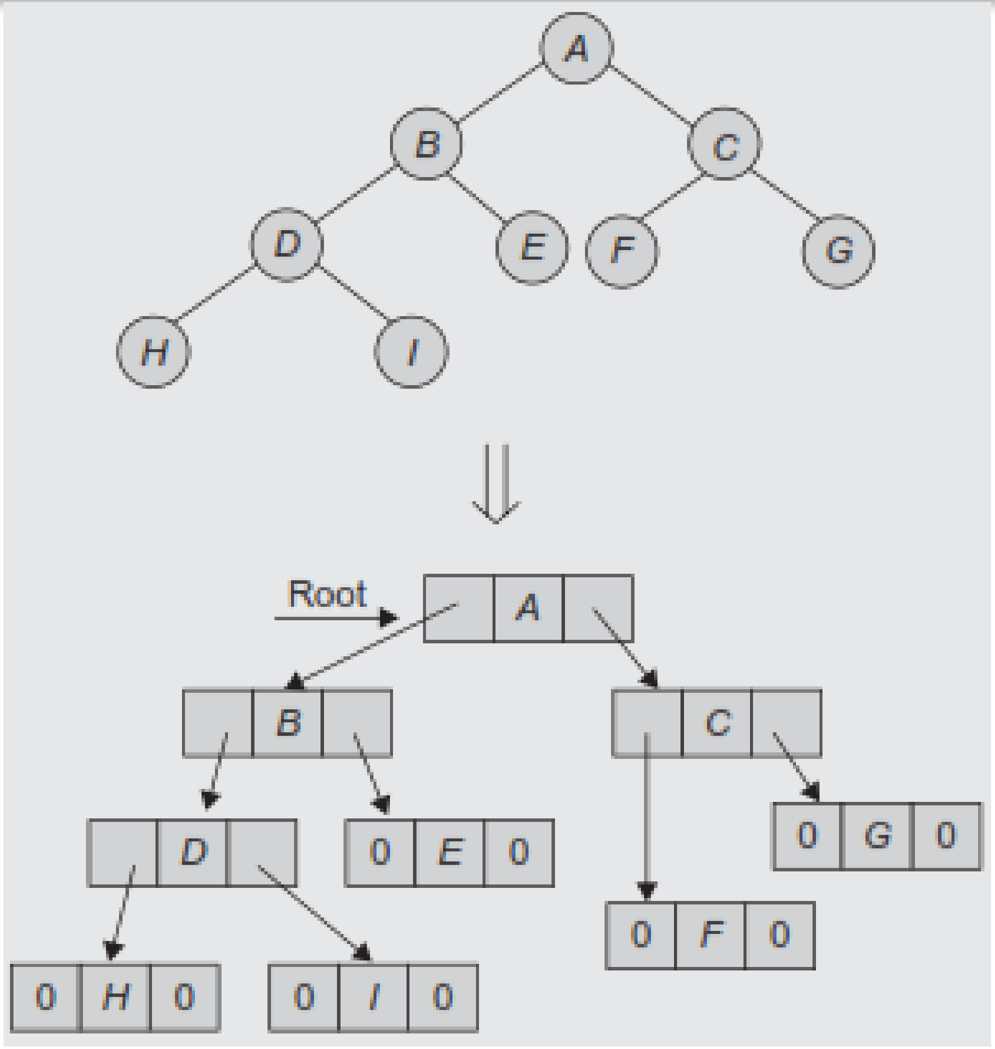
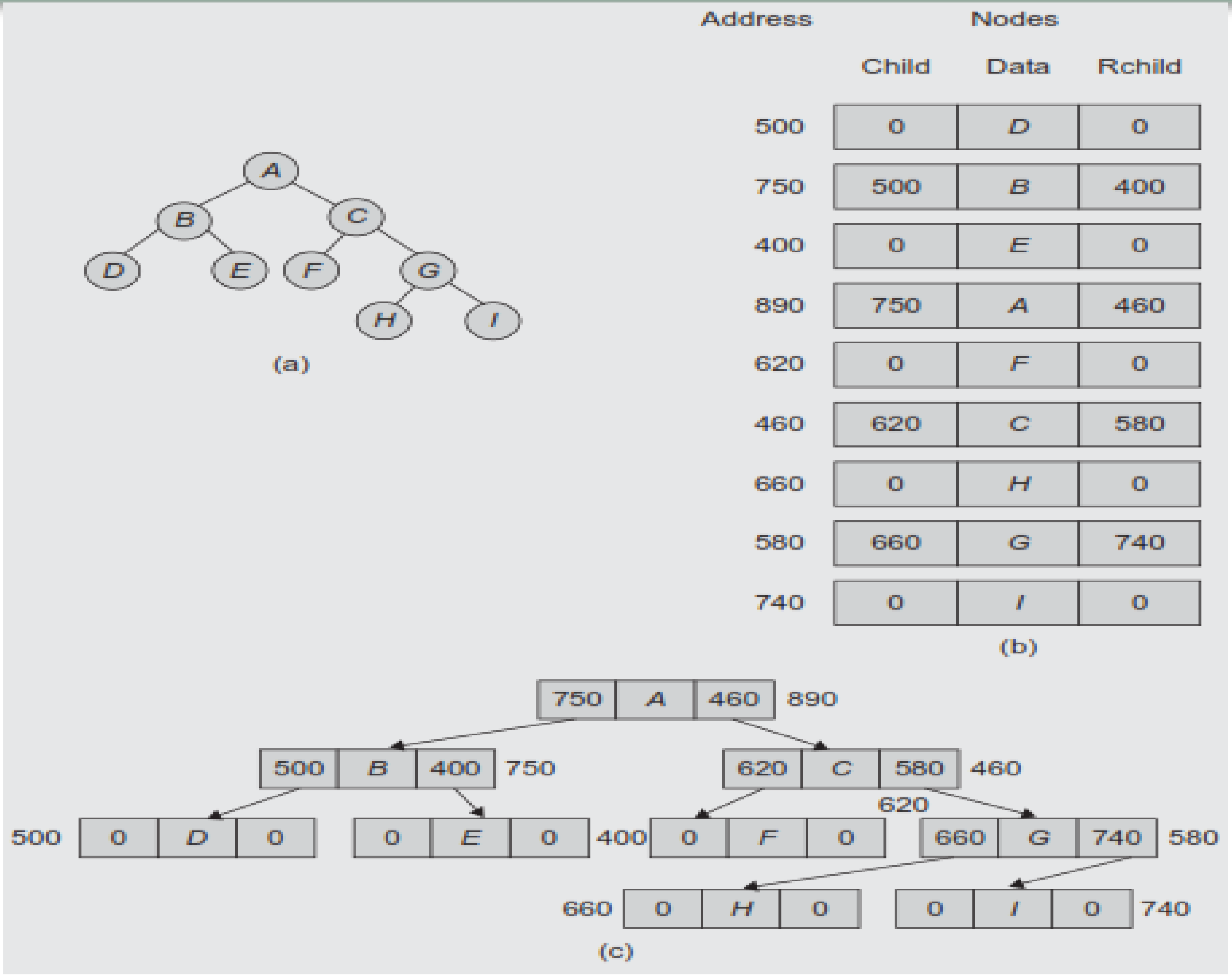


Fig. Sample tree 1 and its linked representation

Linked Representation…

```
class BinaryTree
{
  private:
    TreeNode *Root;
  public:
    BinaryTree() {Root = NULL;}
        TreeNode *GetNode();
    void InsertNode(TreeNode*);
    void DeleteNode(TreeNode*);
    void Postorder(TreeNode*);
    void Inorder(TreeNode*);
    void Preorder(TreeNode*);
};
```

```
class TreeNode
{
  public:
    char Data;
     TreeNode *Lchild;
    TreeNode *Rchild;
};
```

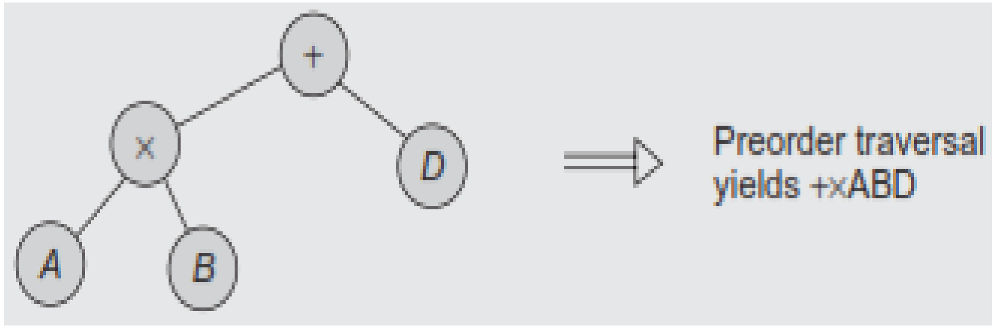


Address	Nodes		
	Child	Data	Rchild
500	0	D	0
750	500	B	400
400	0	E	0
890	750	A	460
620	0	F	0
460	620	C	580
660	0	H	0
580	660	G	740
740	0	I	0

Fig. Tree and its views (a) Binary tree (b) Physical view (c) Logical view

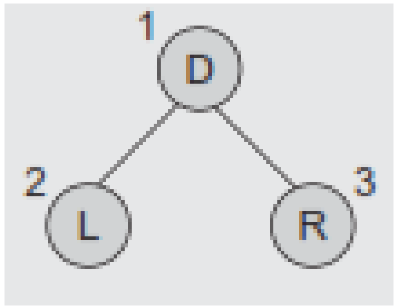
Preorder Traversal

- The root is visited first followed by the left subtree in preorder and then the right subtree in preorder.
- ***Preorder (DLR) Algorithm***
  1. Visit the root node, sayD.
  2. Traverse the left subtree of the node in preorder.
  3. Traverse the right subtree of the node in preorder.



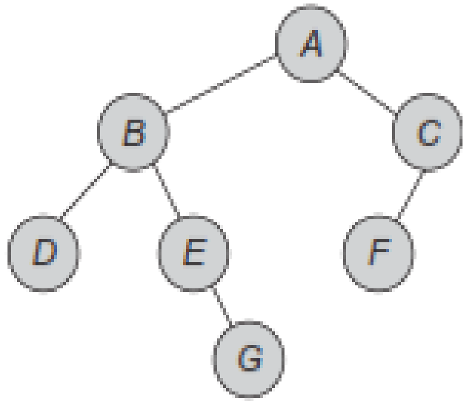
Preorder Traversal…

```
void BinaryTree :: Preorder(TreeNode*)
{
  if(Root != NULL)
  {
    cout << Root->Data;
    Preorder(Root->Lchild);
    Preorder(Root->Rchild);
  }
}
```



Inorder Traversal

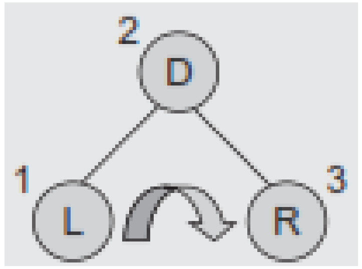
- In this traversal, the left subtree is visited first in inorder followed by the root and then the right subtree in inorder. This can be defined as the following:
- ***Inorder (LDR) Algorithm***
    1. Traverse the left subtree of the root node in inorder.
    2. Visit the root node.
    3. Traverse the right subtree of the root node in inorder.



Inorder sequence: D B E G A F C

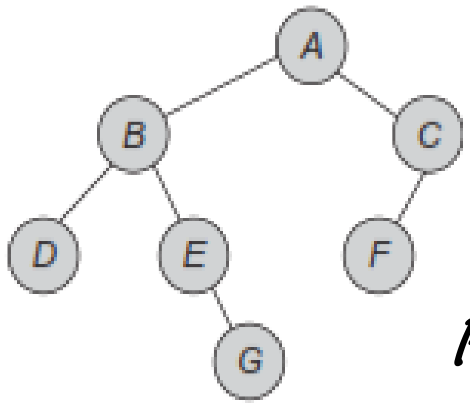
Inorder Traversal…

```
void BinaryTree :: Inorder(TreeNode*)
{
  if(Root  != NULL)
  {
    Inorder(Root → Lchild);
    cout << Root → Data;
    Inorder(Root → Rchild);
  }
}
```



Postorder Traversal

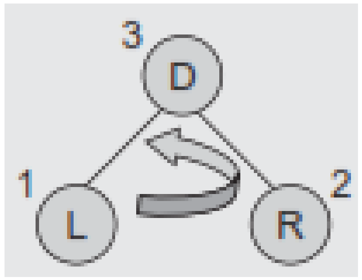
- In this traversal, the left subtree is visited first in postorder followed by the right subtree in postorder and then the root.
- ***Postorder (LRD) Algorithm***
    1. Traverse the root's left child (subtree) of the root node in postorder.
    2. Traverse the root's right child (subtree) of the root node in postorder.
    3. Visit the root node.



Postorder sequence: D G E B F C A

Postorder Traversal...

```
void BinaryTree :: Postorder(TreeNode*)
{
  if(Root != NULL)
  {
    Postorder(Root →Lchild);
    Postorder(Root→Rchild);
    cout << Root→Data;
  }
}
```



Non-recursive Preorder implementation

```
void BinaryTree :: Preorder_Non_Recursive()
{
        TreeNode *Tmp = Root;
        stack S;
        while(1)
        {
                 while(Tmp != NULL)
                // traverse left till left is NULL and push
                {
                        S.Push(Tmp);
                        cout << Tmp ->Data;
                        Tmp = Tmp->Lchild;
                }
                if(S.IsEmpty()) return;
                 //if stack is not empty then pop one and go to right
                Tmp = S.Pop();
                Tmp = Tmp->Rchild;
                // if stack is empty stop the process
        }
}
```

Non-recursive Inorder Traversal

- **Algorithm Inorder**

```
1. Tmp = Root
2. while(1) do
   begin
       while(Tmp != NULL) then
       begin
          push(Tmp)
          Tmp = Tmp→Lchild
       end
       if(stack is empty) then exit
       Tmp = Pop()
       visit(Tmp → data)
       Tmp = Tmp → Rchild
     end while
3. Stop
```

Non-recursive Inorder Traversal

```
void BinaryTree :: Inorder_Non_Recursive()
{
        TreeNode *Tmp;
        stack S;
        Tmp = Root;
       while(1)
       {
                while(Tmp != NULL)
                {
                        S.Push(Tmp);
                        Tmp = Tmp→Lchild;
                }
                if(S.IsEmpty())
                        return;
                  //if stack is not empty then pop one and go to right
                Tmp = S.Pop();
                cout << Tmp→Data;
                Tmp = Tmp→Rchild;
        }
}
```

Non-recursive Postorder Traversal

- As compared to earlier algorithms, in postorder traversal, we require thePop operation when returning from the left and right subtrees.
- **Algorithm**
  1. When we return from the left subtree, perform the following operations:
     a) Tmp = Pop
     b) Tmp = Tmp→Rchild.
  2. When we return from the right subtree, perform the following operations:
     a) Tmp = Pop
     b) Print Tmp→data (i.e, visit and process the node, if required)
     c) Tmp = Pop
- Hence, we need to differentiate between the return operation from the left subtree and right subtree.
- Let us use the stack that stores the status: 'L' for left , and 'R' for right.
  For performing the extra Pop operation while returning from the right subtree, we need to assign Tmp = NULL.

Non-recursive Postorder Traversal

```
void BinaryTree ::
Postorder_Non_Recursive () {
  TreeNode * Tmp = Root;
  stack S;
  stack1 S1;
  char flag;
  // stack S stores the node and
  //S1 stores the flag 'L' or 'R'
  while(1)  {
    while(Tmp != NULL) {
    // traverse tree left till not Null
      S.Push(Tmp); // push node in S
      S1.Push1('L'); // push 'L' in S1
      Tmp = Tmp→Lchild;
    }
    if(S.IsEmpty())
      return;
      else  {
        Tmp = S.Pop();   //pop node
        flag = S1.Pop1();
        if(flag == 'R')  {
         // if flag is 'R' display data
          cout << Tmp→Data;
          Tmp = NULL;
        }
        else {              // if flag is 'L'
            S.Push( Tmp); // push Tmp to S
            S1.Push1('R'); // push 'R' to S1
          Tmp = Tmp→Rchild;  // move to
right
        }
      }
    }
}
```

Formation Of Binary Tree From Its Traversals

- The basic principle for formulation is as follows:
  1. If the preorder traversal is given, then the first node is the root node.
  2. If the postorder traversal is given, then the last node is the root node.
  3. Once the root node is identified, all the nodes in all left and right subtrees of the root node can be identified.
  4. Same techniques can be applied repeatedly to form the subtrees.
- We can conclude that for the binary tree, construction and traversals are essential out of which one should beinorder traversal and another should be preorder or postorder traversal.
- Alternatively, for the given preorder and postorder traversals, the binary tree cannot be obtained uniquely.

Formation Of Binary Tree From…

- Example: Construct a binary tree using the following two traversals:
  Inorder: D B H E A I F J C G
  Preorder: A B D E H C F I J G

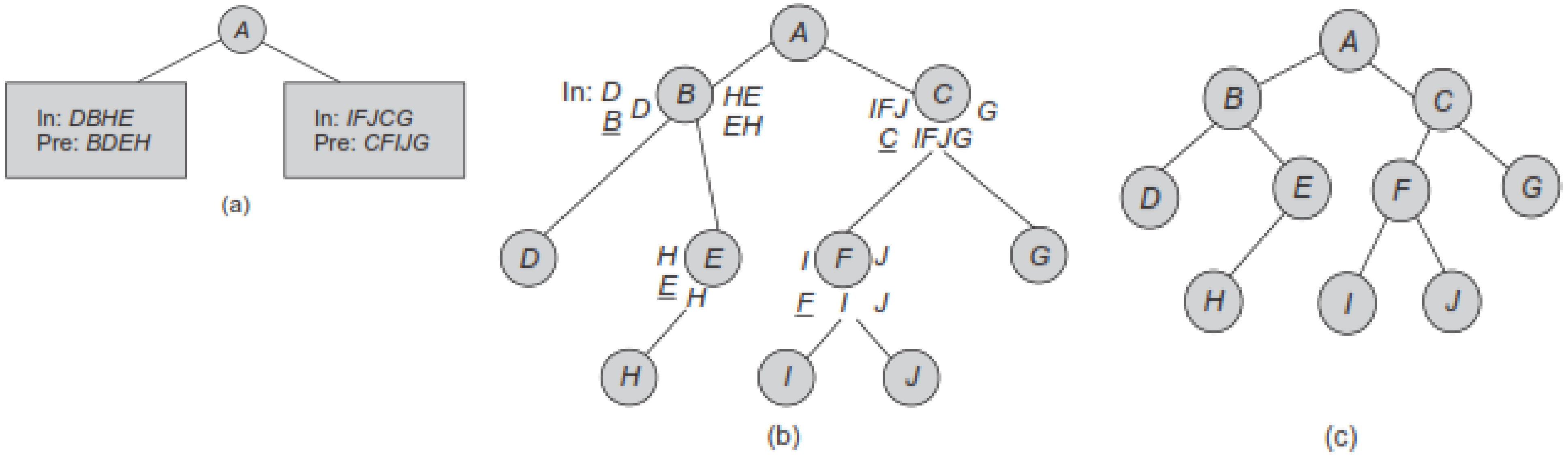


Fig. Binary tree from inorder and preorder traversals (a) Two subtrees as a being the root from two traversals (b) Repeated application (c) Final binary tree

Formation Of Binary Tree From…

- **Example 2** Construct a binary tree from its inorder and postorder traversals.
  Inorder : 1 2 3 4 5 6 7 8 9
  Postorder: 1 3 5 4 2 8 7 9 6

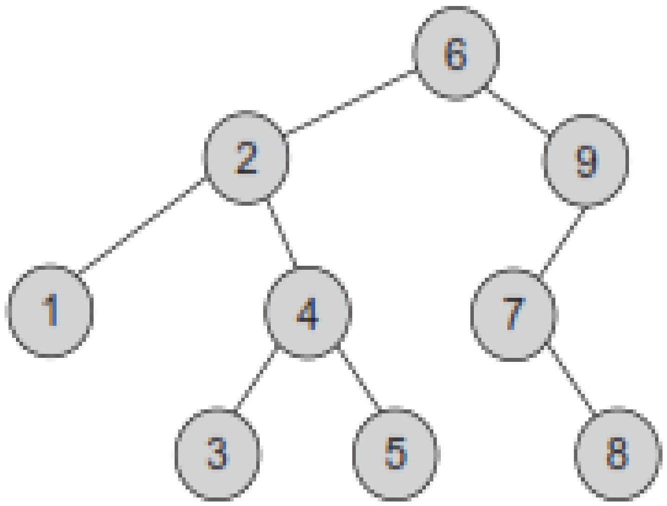


Fig. Final binary tree

Other Tree Operations

- Using traversal as a basic operation, many other operations can be performed on a tree, such as finding the height of the tree, computing the total number of nodes, leaf nodes, and so on.
- **Counting nodes:** returns the total count of nodes in a linked binary tree.

```
int  BinaryTree :: CountNode(TreeNode *Root)
{
  if(Root == NULL)
    return 0;
  else
    return(1 + CountNode(Root->Rchild) +
        CountNode(Root->Lchild));
}
```

Other Tree Operations…

- **Counting leaf nodes:** *Leaf nodes are those with no left or right children*
  - The CountLeaf() operation counts the total number of leaf nodes in a linked binary tree.

```
int BinaryTree :: CountLeaf(TreeNode *Root)
{
    if(Root == NULL)
        return 0;
    else if((Root->Rchild == NULL) && (Root->Lchild == NULL))
        return(1);
    else
        return(CountLeaf(Root->Lchild) + CountLeaf(Root
    ->Rchild));
}
```

Binary Search Tree (BST)

- The binary search tree (BST) is a binary tree with the property that the value in a node is greater than any value in a node's left subtree and less than any value in the node's right subtree.
- A BST is a binary tree that is either empty or where every node contains a key and satisfies the following conditions:
  1. The key in the left child of a node, if it exists, is less than the key in its parent node.
  2. The key in the right child of a node, if it exists, is greater than the key in its parent node.
  3. The left and right subtrees of a node are again BSTs.
- We assume that all keys are unique.
- This property guarantees fast search time provided the tree is relatively balanced.

Binary Search Tree (BST)...

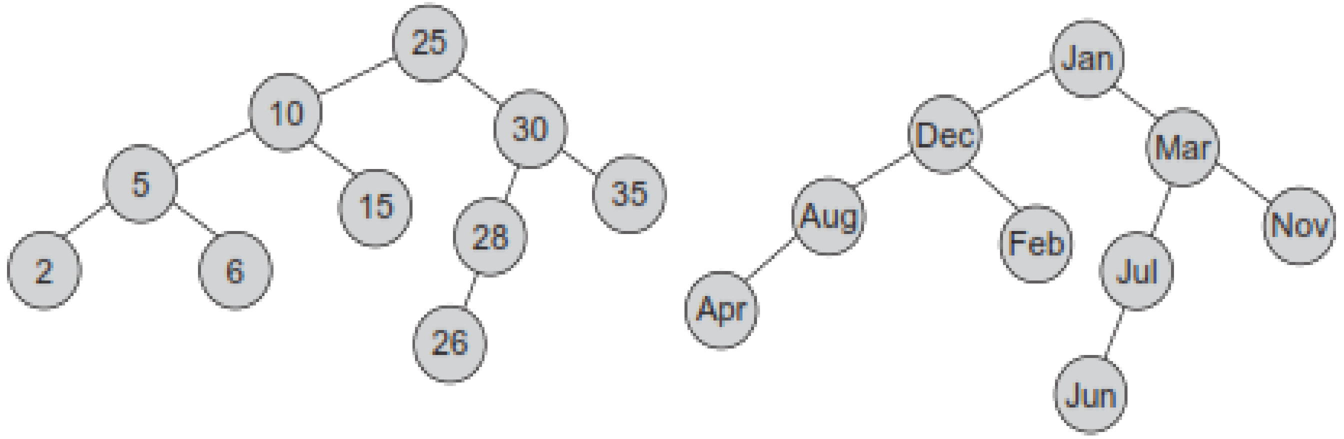


Fig: Binary search trees

- The following are the operations commonly performed on a BST:
  1. Searching a key
  2. Inserting a key
  3. Deleting a key
  4. Traversing the tree

Binary Search Tree (BST)...

```
class BSTree
{
  private:
    TreeNode *Root;
  public:
    BSTree() {Root = NULL;}
    void InsertNode(int Key) ;
    void DeleteNode(int key) ;
    void Search(int Key) ;
    bool IsEmpty() ;
};
```

```
class TreeNode
{
    <data type> Key;
    TreeNode *Lchild;
    TreeNode *Rchild;
};
```

BST: Inserting A Node

- To insert a new node into a BST, the keys should remain in proper order so that the resulting tree satisfies the definition of a BST.
- If the tree is empty, then the first entry when inserted, becomes the root.
- If the tree is not empty, then search the appropriate position and insert the node in that position
- Note that the inorder traversal of a BST generates the data in ascending order.

BST: Inserting A Node…

- ALGORITHM
  - Suppose NewNode is a pointer variable to hold the address of the newly created node. DATA is the information to be pushed.

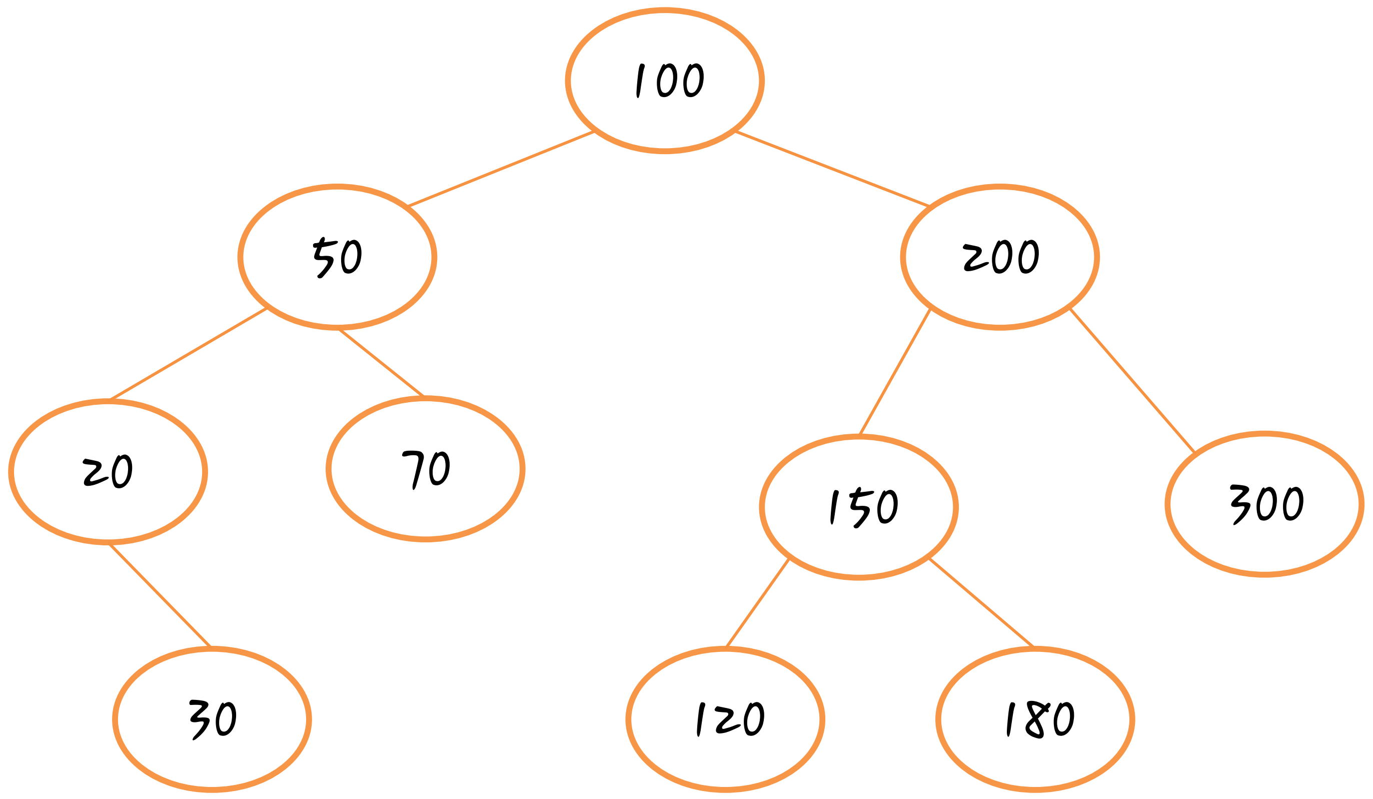
```
1.  Input the DATA to be pushed and ROOT node of the tree.
2.  ParentNode = RootNode
3.  If (ParentNode == NULL) then
        RootNode = NewNode and GoTo Step 7
4.  Else If (NewNode→Data < ParentNode →Data)
    a)  If ParentNode→Left != NULL, then
         ParentNode = ParentNode →Left and GoTo Step 4
    b)  Else
         ParentNode→Left = NewNode and GoTo Step 7
5.  Else If (DATA > ParentNode→ Data)
    a)  If ParentNode→Right != NULL
         ParentNode = ParentNode→Right and then GoTo Step 4
    b)  Else
         ParentNode -> Right = NewNode and GoTo Step 7
6.  Else
        Display ("DUPLICATE NODE")
7.  EXIT
```

BST: Inserting A Node…

```
void BSTree :: Insert(int Key)  {
    TreeNode *Tmp, NewNode;
    NewNode = new BSTNode;
    NewNode->Data = Key;
    NewNode->Lchild = NewNode->Rchild
    = NULL;
    if (Root == NULL)    {
        Root = NewNode;
        return;
    }
    Tmp = Root;
    While(1) {
        if(Tmp→Data > NewNode→Data)
        {
            if(Tmp→Left==NULL) {
                Tmp->Left = NewNode;
                return;
            }
            else
                Temp = Tmp→Left;
        }
        else  {
            if(Tmp→Right==NULL){
                Tmp→Right = NewNode;
                return;
            }
            else
                Tmp = Tmp->Right;
        }
    }
}
```

BST: Inserting A Node

- **Example 7.6** Build a BST from the following set of elements—100, 50, 200, 300, 20, 150, 70, 180, 120, 30.



Searching For A Key in BST

- To search for a target key, we first compare it with the key at the root of the tree.
  - If it is the same, then the algorithm ends.
  - If it is less than the key at the root, search for the target key in the left subtree,
  - else search in the right subtree.

Searching For A Key in BST

```
TreeNode *BSTree :: Search(int Key)
{
  TreeNode *Tmp = Root;
  while(Tmp)   {
    if(Tmp->Data == Key)
      return Tmp;
    else if(Tmp->data < Key)
      Tmp = Tmp->Lchild;
    else
      Tmp = Tmp->Rchild;
  }
  return NULL;
}
```

```
TreeNode *BSTree ::
Rec_Search(TreeNode *root, int key)
{
  if(root == NULL)
    return(root);
  else  {
    if(root->Data < Key)
      root = Rec_Search(root
->Lchild);
    else if(root->data > Key)
      root = Rec_Search(root
->Rchild);
  }
}
```

Deleting a node in BST

- Deletion of a node is one of the frequently performed operations.
- Let **T** be a BST and **X** be the node of key **K** to be deleted from T, if it exists in the tree.
- Let **Y** be a parent node of **X**.
- There are three cases when a node is to be deleted from a BST. Let us consider each case:
    1. X is a leaf node.
    2. X has one child.
    3. X has both child nodes.

Deleting a node…

- **Case 1: Leaf node deletion:**
  - If the node to be deleted, say X, is a leaf node, then the process is easy.
  - We need to change the child link of the parent node, say Y of node to be deleted to NULL, and free the memory occupied by the node to be deleted and then return.
  - Consider the following tree given below. Here, 5 is the node to b

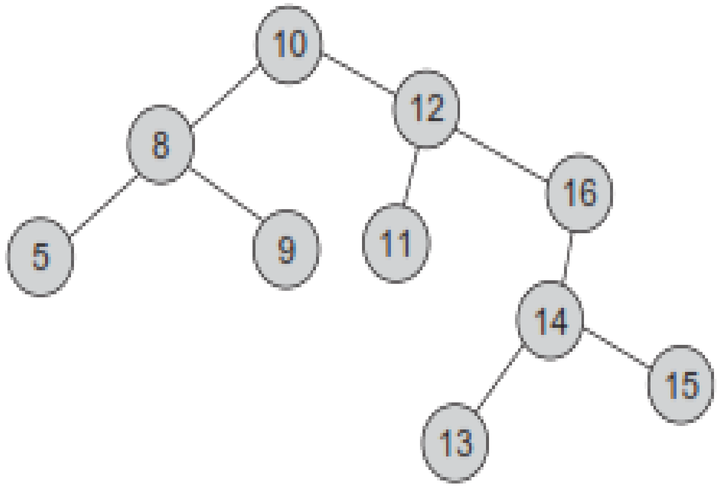


Fig: BST Before deletion of node with data = 5

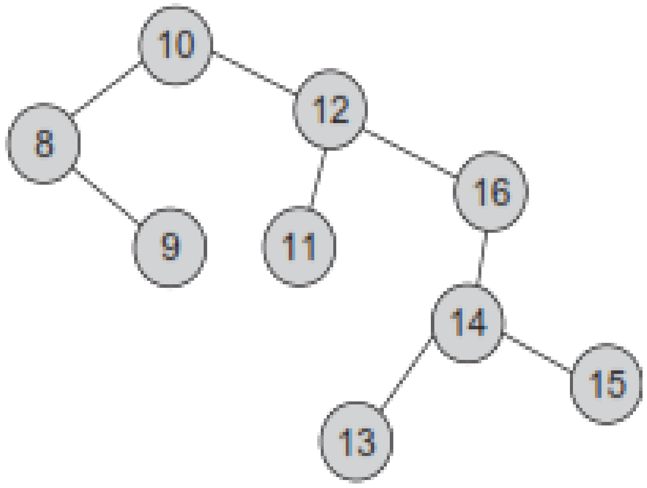


Fig: BST after deletion of node with data = 5

Deleting a node…

- **Case 2(a): Node not having right subtree**
  - If the node to be deleted has a single child link, that is, either right child or left child is NULL and has only one subtree, the process is still easy.
  - If there is no right subtree, then just link the left subtree of the node to be deleted to its parent and free its memory.
  - If X denotes the node to be deleted and Y is its parent with X as a left child, then we need to set Y->Lchild = X ->Lchild and free the memory.
  - If X denotes the node to be deleted and Y is its parent with X as a right child, then we need to set Y->Rchild =X ->Lchild and free the memory.

Deleting a node...

- **Case 2(b) : Node not having left subtree**
  - If there is no left subtree, then just link the right subtree of the node to be deleted to its parent and free its memory.
  - If X denotes the node to be deleted and Y is its parent with X as a left child, then we need to set Y->Lchild = X ->Rchild and free the memory.
  - If X denotes the node to be deleted and Y is its parent with X as a right child, then we need to set Y->Rchild = X ->Rchild and free the memory.

Deleting a node…

- **Case 3: Node having both subtrees**
- Approach 1: Deletion by merging – one of the following is done
  - If the deleted node is the left child of its parent, one of the following is done
    - The left child of the deleted node is made the left child of the parent of the deleted node,
    - The right child of the deleted node is made the right child of the node containing largest element in the left of the deleted node
  - OR
    - The right child of the deleted node is made the left child of the parent of the deleted node,
    - The left child of the deleted node is made the left child of the node containing smallest element in the right of the deleted node

Deleting a node...

- If the deleted node is the right child of its parent, one of the following is done
  - The left child of the deleted node is made the right child of the parent of the deleted node,
  - The right child of the deleted node is made the right child of the node containing largest element in the left of the deleted node
- OR
  - The right child of the deleted node is made the right child of the parent of the deleted node,
  - The left child of the deleted node is made the left child of the node containing smallest element in the right of the deleted node

Deleting a node…

- Approach 2: Deletion by copying– the following is done
  - Copy the node containing the largest element in the left (or the smallest element in the right) to the node containing the element to be deleted
  - Delete the copied node

Deleting a node…

```
// Function to delete a node
from BST
TreeNode *BSTree :: del(int
deldata) {
   TreeNode *temp = Root,
*parent, *x;
  if(Root == NULL)  {
     cout << endl << "\t BST is
empty";
     return NULL;
  }
  else  {
     parent = temp;
    //Search a BST node to be
          //deleted & its parent
while(temp != NULL)  {
  if(temp->Data == deldata )
    break;     // found
  if(deldata < temp->Data)  {
    parent = temp;
    temp = temp->Lchild;
  }
  else  {
    parent = temp;
    temp = temp->Rchild;
  }
}     // end of search
if(temp == NULL)
  return(NULL);
```

Deleting a node…

```
else {
  //BST node having right
children
  if(temp->Rchild != NULL) {
   //Temp is having right child";
    parent = temp;
    x = temp->Rchild;
    while(x->Lchild != NULL)  {
      parent = x;
      x = x->Lchild;
    }
    temp->Data = x->Data;
    temp = x;
  }
```

```
        //BST node being a leaf Node
     if(temp->Lchild == NULL &&
temp->Rchild == NULL)   {
        //Leaf node;
       if(temp != root)  {
         if(parent->lLchild ==
temp)
           parent->Rchild = NULL;
         else
           parent->Rchild = NULL;
       }
       else
         root = NULL;
       delete temp;
       return(root);
     }
     else if(temp->Lchild !=NULL
&&     temp->Rchild == NULL)
```

Deleting a node…

```
  else if(temp->Lchild !=NULL &&   temp->Rchild == NULL)
   {// BST node having left children
     if(temp != root)   {
         if(parent->Lchild == temp)
             parent->Lchild = temp->Lchild;
         else
             parent->Rchild = temp->Lchild;
     }
     else
         root = temp->Lchild;
     delete temp;
     return(root) ;
    }
   }
  }
}
```

Applications Of Binary Trees

- There is a vast set of applications of the binary tree in addition to searching.
- The applications discussed in this section are gaming, expression tree, Huffman tree for coding, and decision trees.
- Let us see expression trees here in this chapter.

Expression Tree

- A binary tree storing or representing an arithmetic expression is called asexpression tree.
- The leaves of an expression tree areoperands. Operands could be variables or constants.
- The branch nodes (internal nodes) represent the operators.
- A binary tree is the most suitable one for arithmetic expressions as it contains either binary or unary operators.

  LetE = $((A \times B) + (C - D)) / (C - E)$

Pre order traversal – generates expression in prefix notation.
Inorder traversal – generates expression in infix notation.
Postorder traversal – generates expression in postfix notation.

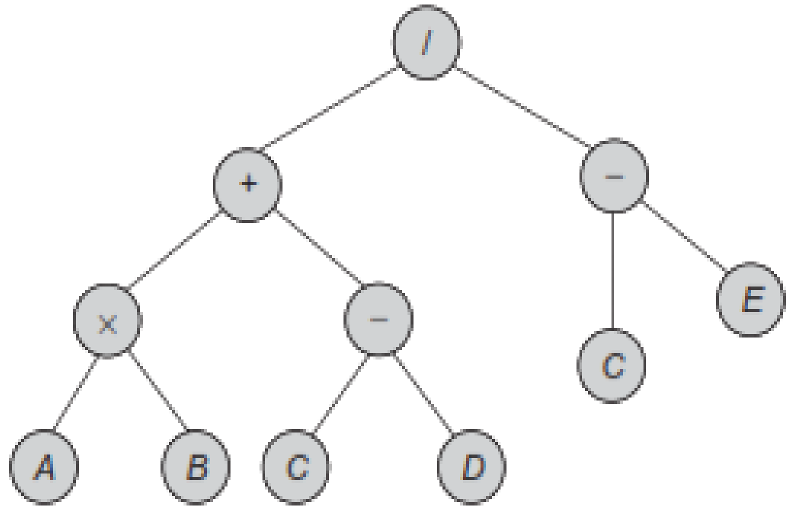


Fig. The expression tree for expression E

END OF CHAPTER