Chapter 6- Trees

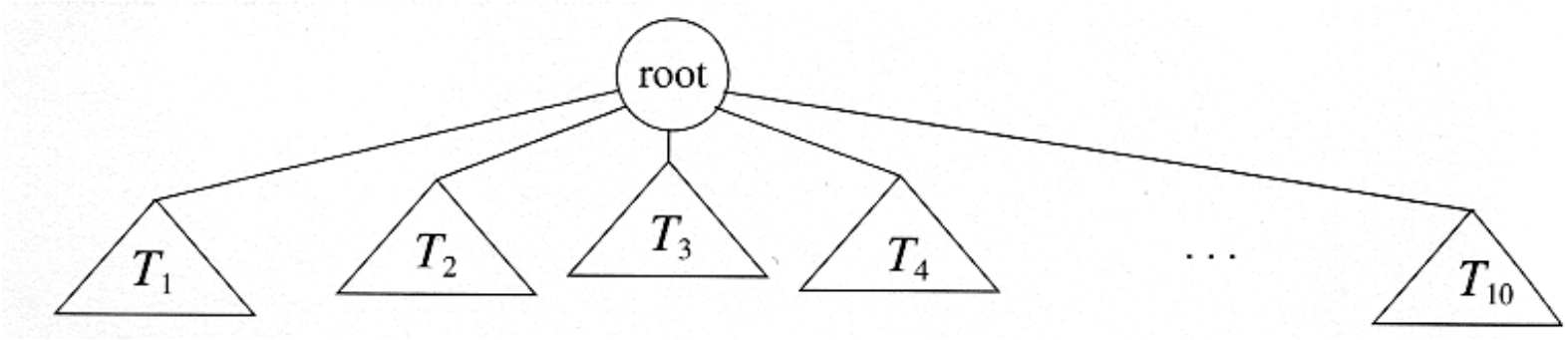
Bekele M.

Trees

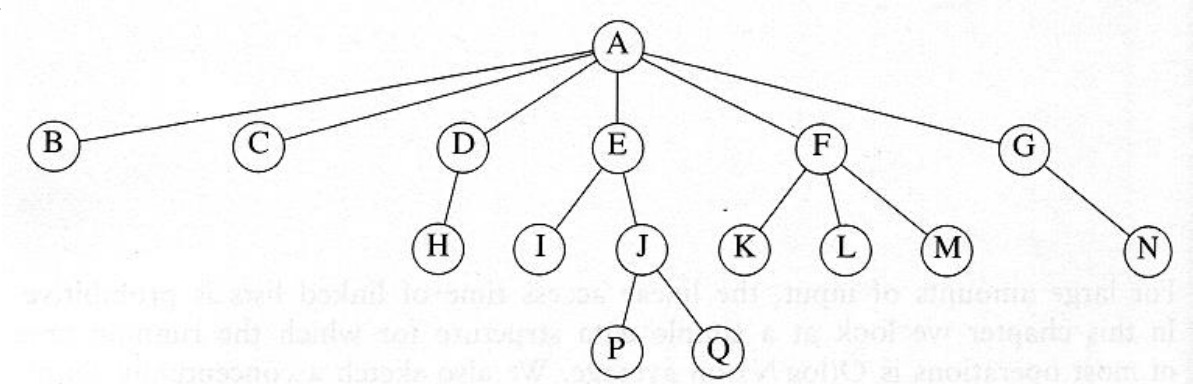
- Linear access time of linked lists and arrays is expensive
  - Requires O(N) running time for most of basic operations like search, insert and delete
- Does there exist any simple data structure for which the running time of most operations (search, insert, delete) is O(log N)?
  - The answer is yes, tree data structures require O(log N) for most of these operations
  - However, unlike Array and Linked List, which are linear data structures, tree is hierarchical (or non-linear) data structure.

Trees

- A tree is a collection of nodes
  - The collection can be empty
- (recursive definition) If not empty, a tree consists of a distinguished node r (the *root*), and zero or more nonempty *sub-trees* $T_1, T_2, ..., T_k$, each of whose roots are connected by an *edge* from r. Each sub tree itself satisfies the definition of a tree.

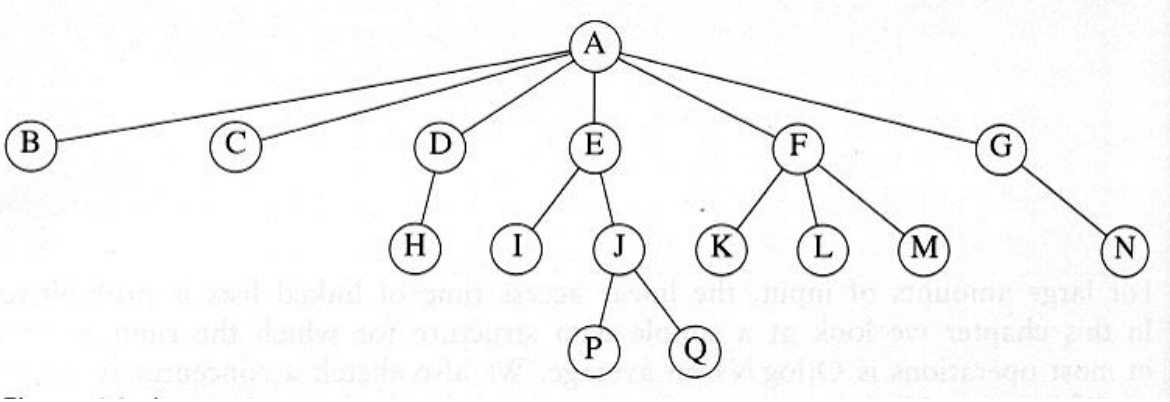


Some Tree Terminologies



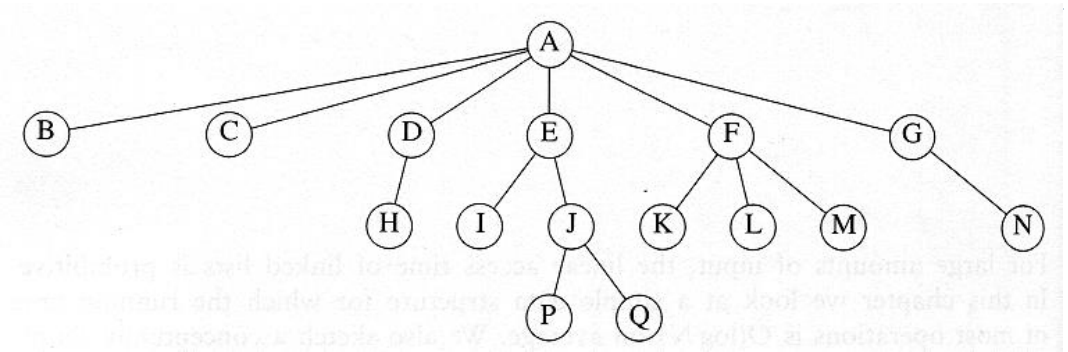
- *Child* and *parent*
  - Every node except the root has one parent (J is a parent of P and Q)
  - **P** and **Q** are child of **J**
  - A node can have an arbitrary number of children (A has 6 while D has 1 child)
- *Leaves/External Nodes*
  - Nodes with no children (B, C, H, I, P, Q, K, L, M, N)
- *Sibling*
  - nodes with same parent (Example, P and Q)
- *Internal node*
  - A node with at least one child (A,D,E,F,G,J)
- *Degree*: the number of possible child of a node
  - Degree of node F = 3, node N=0, degree of the tree=degree of node having maximum degree. (=6)

Some Terminologies



- *Path*
  - *is a sequence of nodes from root to a node (arbitrary node in the tree).*
- *Length*
  - Number of edges on the path from node x to node y
- *Depth* of a node
  - Number of edges from the root to that node (Depth of C =1, of A = 0)
  - The depth of a tree is equal to the depth of the deepest leaf (=3)
    - Because, the deepest are P and Q and their depth is 3

Some Terminologies



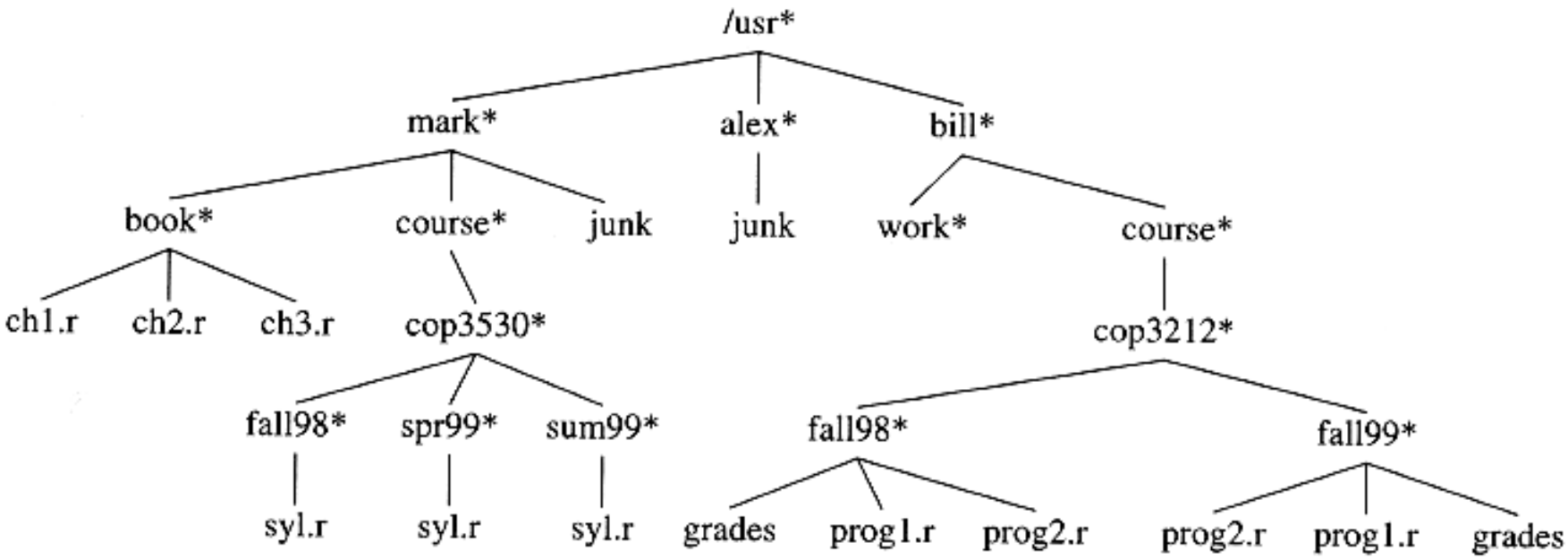
- *Level*
  - *Level of node n, is the depth of node n.*
  - *The level of a node is one greater than the level of its parent.*
- *Height* of a node
  - length of the longest path from that node to a leaf (E=2)
  - all leaves are at height 0
  - The height of a tree is equal to the height of the root
- *Ancestor* and *descendant*
  - The ancestors of a node are all the nodes along the path from the root to the node.
    - Parent, grand parent and great grand parents of anode
  - *Descendant* node reachable by repeated proceeding from parent to child.
    - *Child, grand child and great grand child of a node*

Applications of Trees

- Some of tree applications are:
  - Implementing the file system of several operating systems
  - Evaluation of arithmetic expressions
  - Reduction of the time complexity of some searching operations
  - Storing hierarchies in organizations
  - Efficient sorting
  - Used for representing other data structres like Set ADT

Example: UNIX Directory

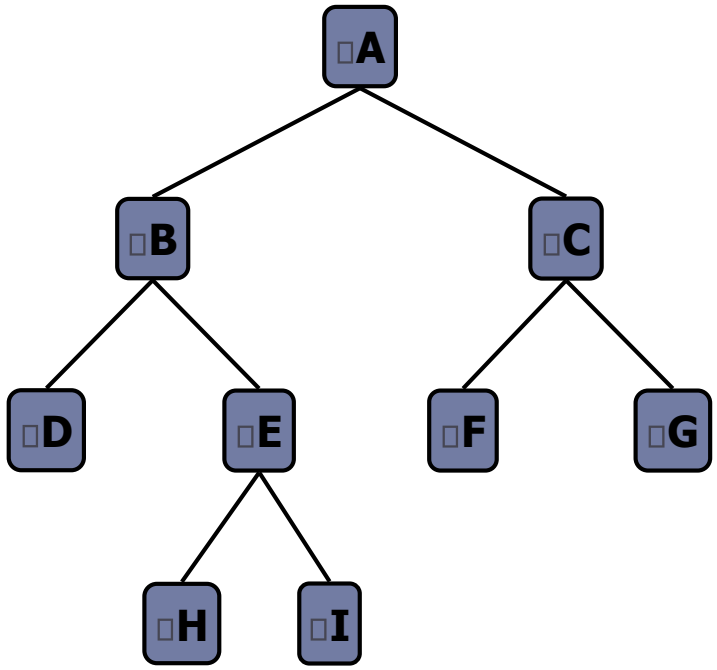
- Tree is useful to represent hierarchical data
- One of its application a file system used by many systems
- The following is an exmple of unix file system



Binary Tree

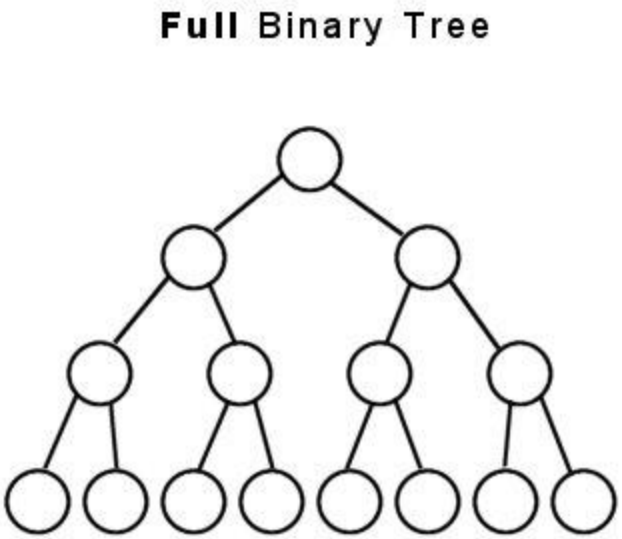
- A binary tree is a tree in which no node can have more than two children
  - Each internal node has at most two children (degree of two)
  - We call the children of an internal node left child and right child

- **Applications:**
  - **arithmetic expressions**
  - **decision processes**
  - **searching**



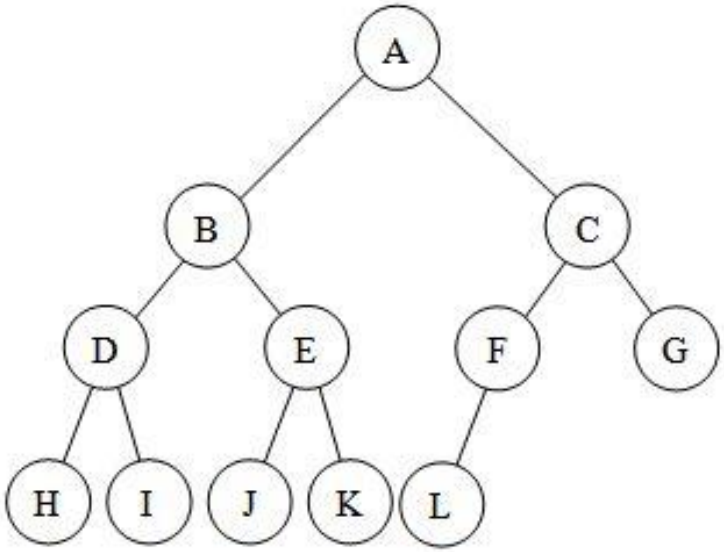
Full binary tree

- **It** is a binary tree in which every node except the leaf nodes has two children
  - a binary tree which has the maximum number of nodes.
  - The following is a full binary tree with depth 3.



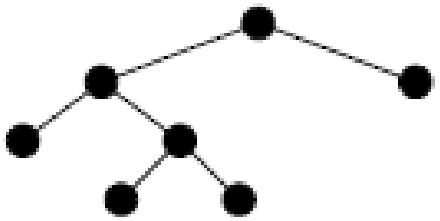
Complete Binary Tree

- A binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible
  - Nodes are filled from left to right in each level

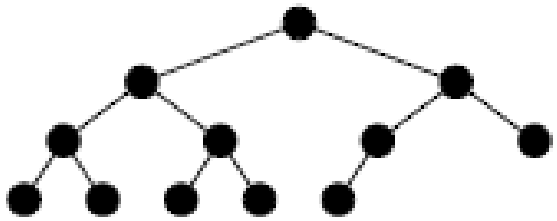


Exercise: Say True/False Based on figure a and b

- Both are binary trees
- Both are full binary trees
- Both are complete binary trees
- A) is full binary while B) is complete binary tree
- B) is full binary while A) is complete binary tree



(a)

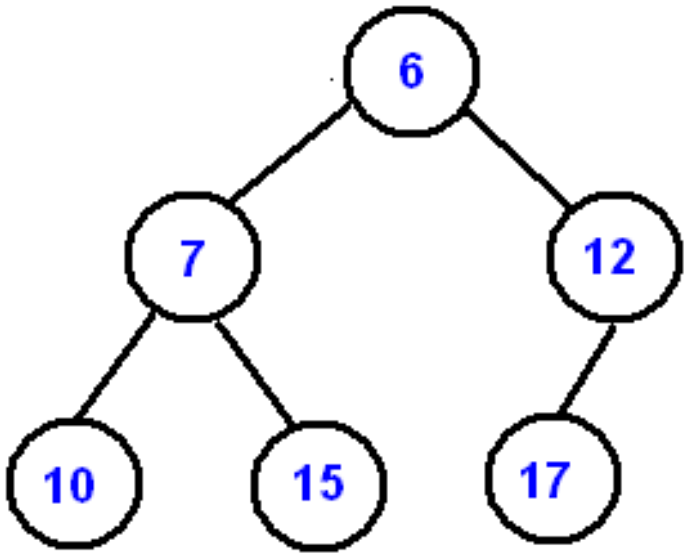


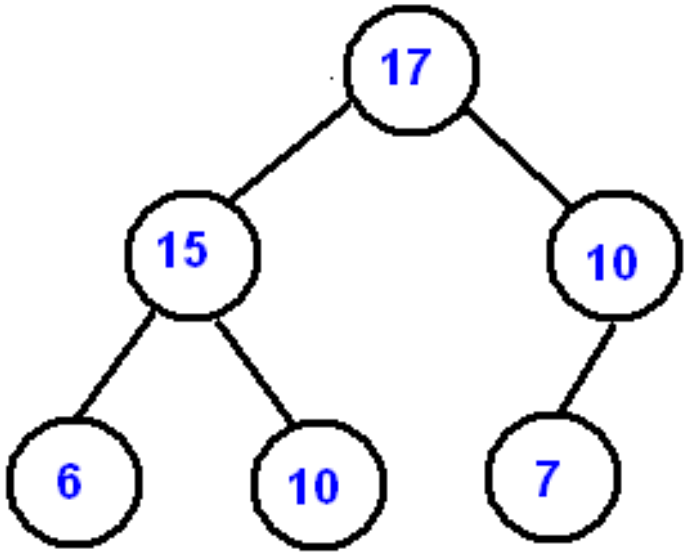
(b)

binary heap (heap)

- A binary heap is **a complete binary tree** which satisfies the heap ordering property. The ordering can be one of two types:
  - The ***min-heap property***: the value of each node is greater than or equal to the value of its parent,
    - the minimum-value element at the root.
  - The ***max-heap property***: the value of each node is less than or equal to the value of its parent,
    - the maximum-value element at the root.
- A common use of a heap is to implement a priority queue
  - In a heap the highest (or lowest) priority element is always stored at the root

Example



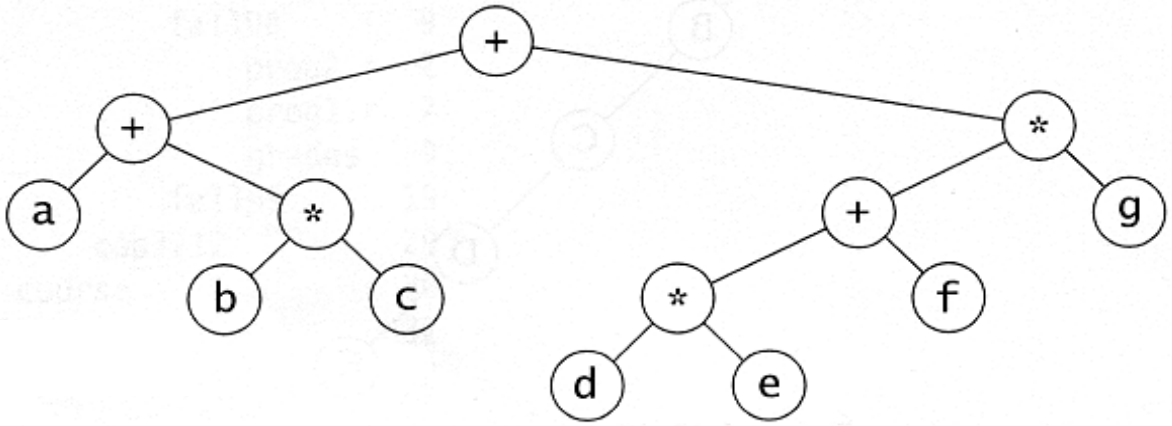


- Min-heap

- Max-heap

Expression Trees

- One of the application of binary tree is representing arithmetic expression
- The hierarchical relationship of parent to children in this tree is that of algebraic operator to its two operands. The operand may itself be an expression (that is, a sub tree) which must be evaluated before the operator in the parent node can be applied.



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

- Leaves are operands (constants or variables)
- The other nodes (internal nodes) contain operators
- Will not be a binary tree if some operators are not binary

Steps to create an expression tree for a given infix expression

1. Convert it to postfix form (Note the algorithm from last chapter)
2. Scan the postfix form character by character starting from the left
3. if you scanned an operand push it to a stack
4. if you scanned an operator,
   A. pop the two top items from the stack and create a binary tree in such a way that the scanned operator will be in the root node, the first popped item will be the right sub tree, and the second popped item will be on the left sub tree.
   B. Then push a pointer to the binary tree into the stack (this will be used as an operand latter)
5. while there is still a character in the input string, scan the next character and repeat steps 3 and 4.

Exercise:

Build an expression tree for:

i). (A-B)*C

ii). (A-B)+C*(E/F)

Tree traversal

- Traversing a tree means processing it in such a way that each node is visited for processing only once.
- There are three traversals methods used to visit/print out the node/data in a tree in a certain order
- Pre-order traversal
  - Print the data at the root
  - Recursively print out all data in the left subtree
  - Recursively print out all data in the right subtree
- In-order Traversal
  - Recursively print out all data in the left subtree
  - Print the data at the root
  - Recursively print out all data in the right subtree
- Post-order Traversal
  - Recursively print out all data in the left subtree
  - Recursively print out all data in the right subtree
  - Print the data at the root

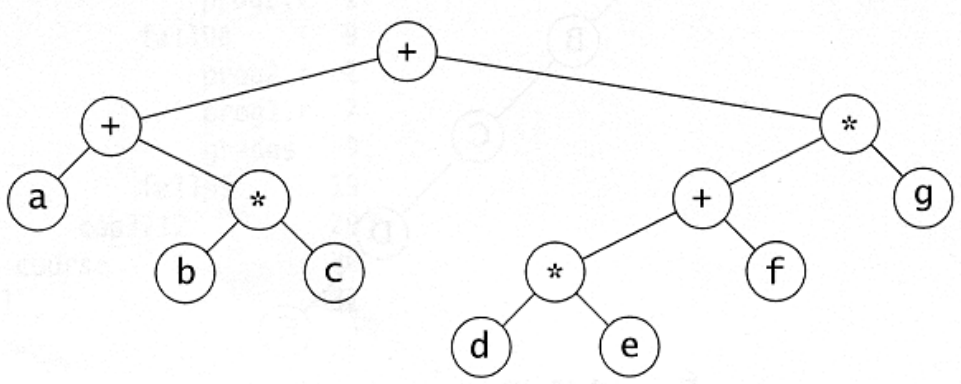
More on Tree traversal

- You may 'pass through' a node as many times as you like but you can only process the node once.

- During a **pre-order traversal** each node is processed before any nodes in its subtrees
- During an **in-order traversal** each node is processed after all nodes in its left subtree but before any nodes in its right subtree.
- During a **post-order traversal** each node is processed after all nodes in both its subtrees

Preorder, Postorder and Inorder

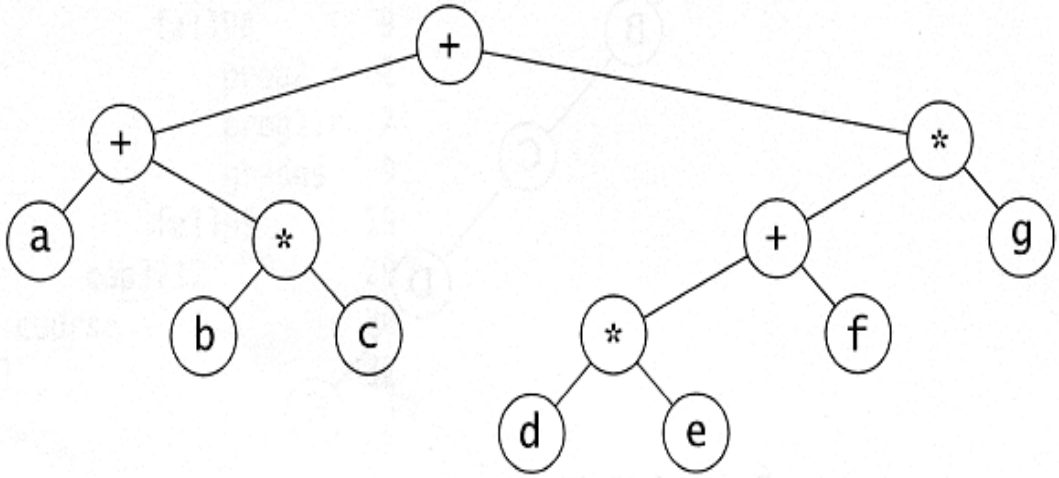
- Once the expression tree is built and if we know exactly how the corresponding tree should be traversed.
  - all the three forms of an algebraic expression (infix, prefix, and postfix) are immediately available to us
- Preorder traversal
  - node, left, right (recursively)
  - It produces prefix expression
    - ++a*bc*+*defg



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

Preorder, Postorder and Inorder

- Post-order traversal
  - left, right, node (recursively)
  - Gives postfix expression
    - abc*+de*f+g*+
- In-order traversal
  - left, node, right. (recursively)
  - Gives infix expression
    - a+b*c+d*e+f*g

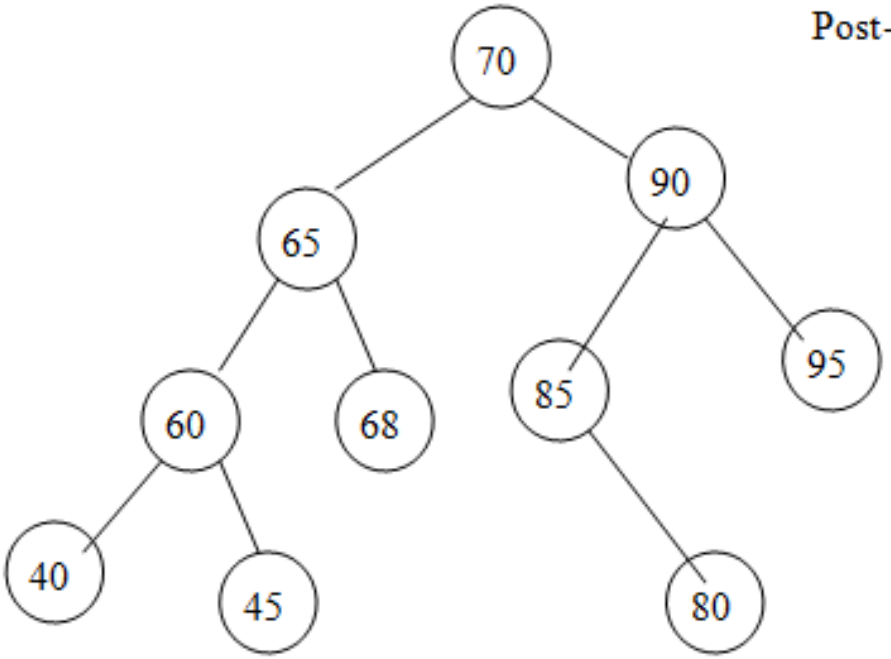


Expression tree for (a + b * c) + ((d * e + f) * g)

Exercise

- Show list of nodes when the following tree is traversed in:

Pre-order: _____
In-order: _____
Post-order: _____

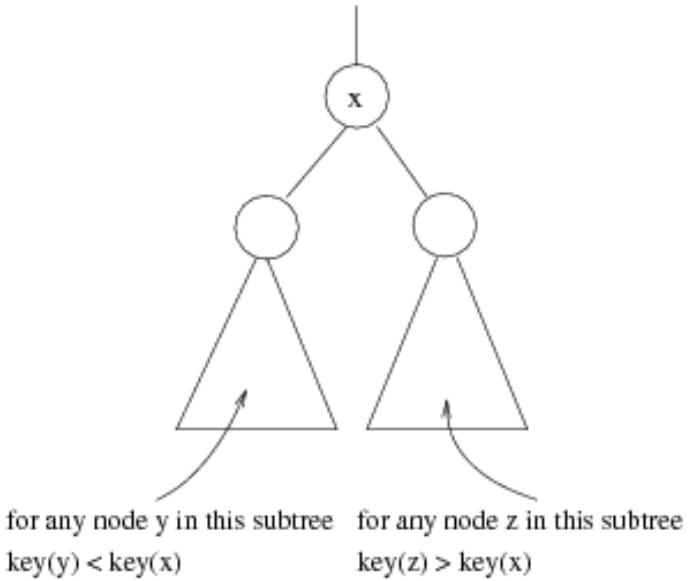


Binary Search Trees

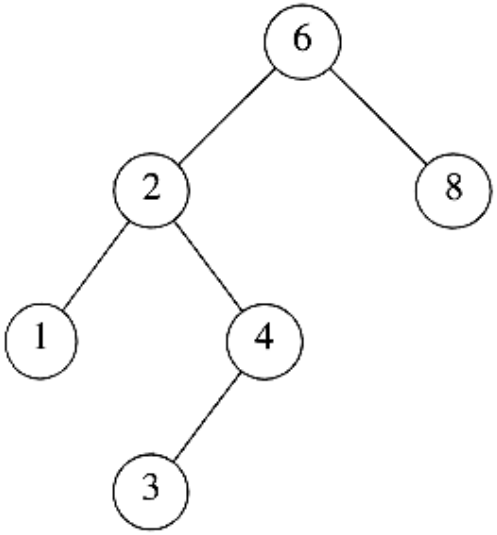
- Stores keys/values in the nodes in a way so that searching, insertion and deletion can be done efficiently.

Binary search tree property

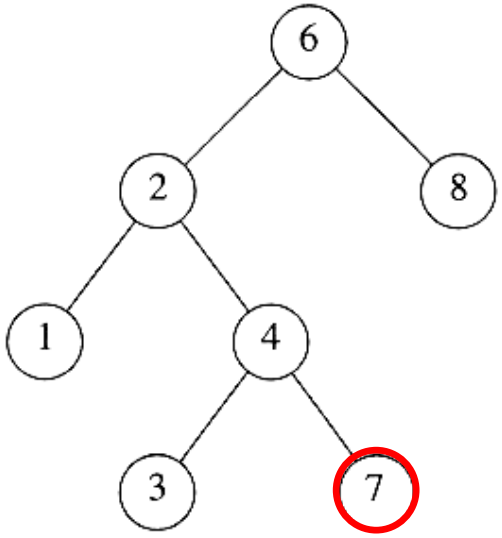
- For every node X, all the keys in its left subtree are smaller than the key value in X, and all the keys in its right subtree are larger than the key value in X



Binary Search Trees



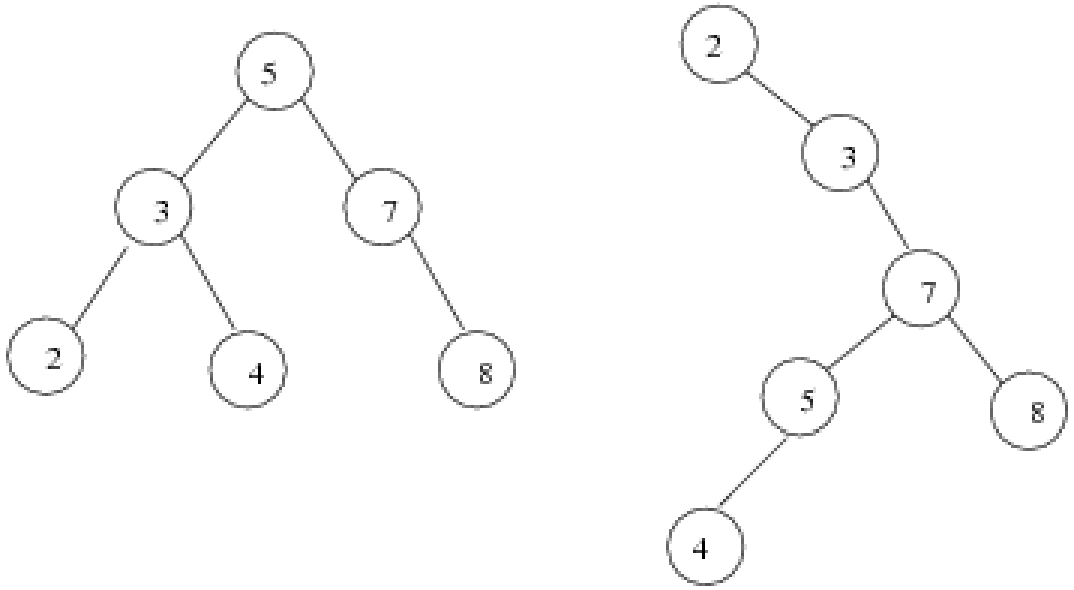
A binary search tree



Not a binary search tree

Binary search trees

Two binary search trees representing the same set:



- Average depth of a node is O(log N); maximum depth of a node is O(N)

Linked representation of binary trees

- Because each node in a binary tree may have two child nodes, a node in a linked representation has two pointer fields
- We will use the following general structure specification

```
struct Node
 {
     int data; //the data type can be any appropriate type
     node  *left;
     node  *right;
 };
     Node *root = NULL; // The root node of the tree
```

Operations on Binary tree

Operation	Explanation
create(root)	Initializes the tree referenced by root
empty(root)	Boolean valued function to determine whether the tree referenced by root is empty
search(root, target)	Searches for a target value on the tree referenced by root
addnode(root, node)	Add to the tree referenced by root the data referenced by node(while maintaining the appropriate hierarchical relationship within the tree)
deletenode(root, node)	Delete from the tree referenced by root the data referenced by node (while maintaining the appropriate hierarchical relationships within the tree)
traverse(root)	In the tree referenced by root, visit every node in specified order relative to the hierarchical relationships within the tree.

```
//
void create( Node &root)
{
 root=NULL;
}
```

```
//
Node get_node()
{
 return new node;
}
```

```
//
boolean empty(pnode root)
{
 return (root==NULL?true:false);
}
```

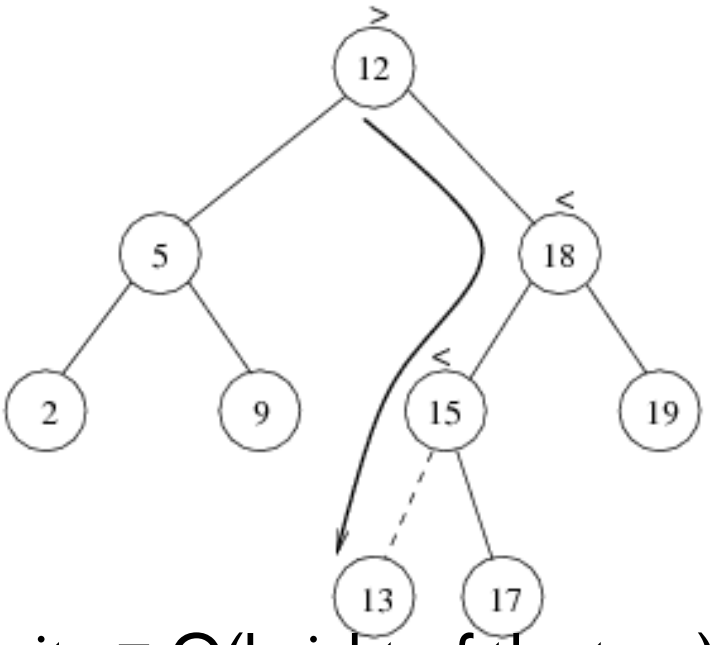
```
//
void free_node(Node &p)
{
 delete p;
}
```

Inserting node in BST

- When a new node is inserted the definition of BST should be preserved.
- Unlike the steps required by a linked list representation,
  - We do not have to traverse the list sequentially to determine where the new node belongs.
  - Instead, we need merely traverse one branch of the tree to determine the position for the new node
- There are two cases to consider
  - There is no data in the tree (root=null)
    - root=newnode;
  - There is data
    - Search the appropriate position
    - Insert the node in that position.

Example- insert node13

- Proceed down the tree as you would with a find
- If newnode is found, do nothing (or update something)
- Otherwise, insert newnode at the last spot on the path traversed



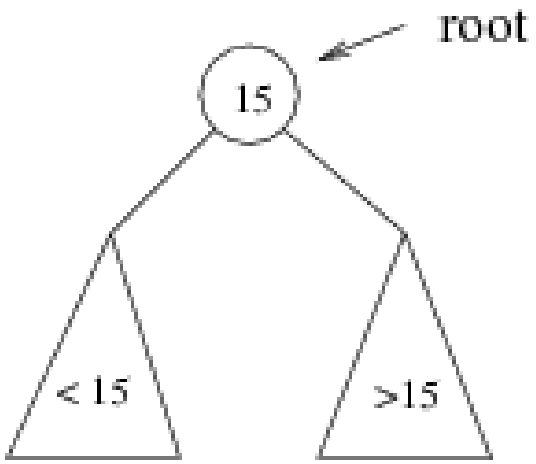
- Time complexity = O(height of the tree)

Add node

```
void add( Node &root, pnode p)
{
 if(root==NULL)
  {
        root=p;
        root->left=NULL;
        root->right=NULL;
  }
  else if(root->data>p->data)
        add(root->left,p);
     else
        add(root->right,p);
}
```

Searching BST

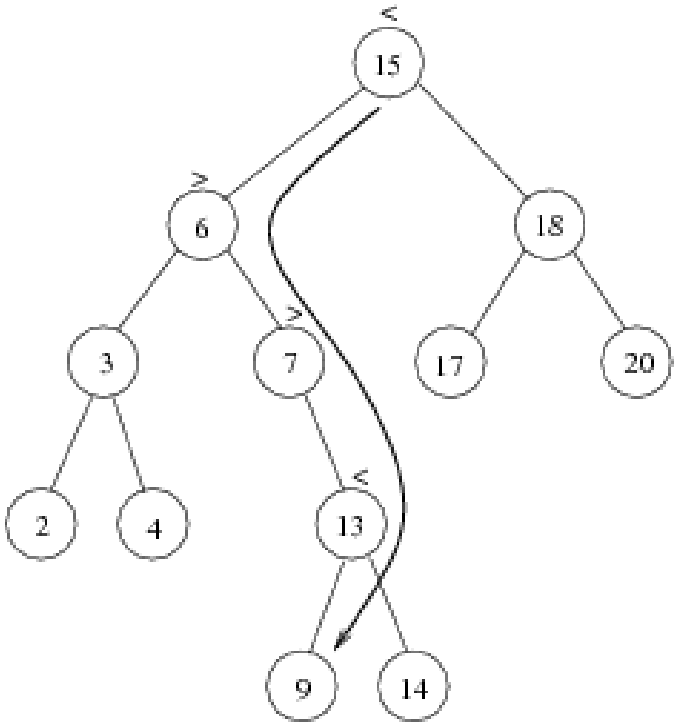
- If we are searching for 15, then we are done.
- If we are searching for a key < 15, then we should search in the left subtree.
- If we are searching for a key > 15, then we should search in the right subtree.



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!



Searching (Find)

- Find X: return a pointer to the node that has key X, or NULL if there is no such node

- Node *searchBST(Node *root, int x)

```
{
   If(root==NULL || root->num==x)
      Return (root)
   else if(root->num>x)
      Return (searchBST(root->left, x)
   else
      Return (searchBST(root->right, x))
      }
```

- Time complexity
  - O(height of the tree)

Tree traversal

```
void pre_order(pnode root)
{
 if(root!=NULL)
 {
  cout<<root->data<<" ";
  pre_order(root->left);
  pre_order(root->right);
 }
 }
```

```
void post_order(pnode root)
{
 if(root!=NULL)
 {
 post_order(root->left);
 post_order(root->right);
 cout<<root->data<<" ";
 }
 }
```

```
void in_order(pnode root)
{
 if(root!=NULL)
 {
 in_order(root->left);
 cout<<root->data<<" ";
 in_order(root->right);
 }
 }
```

findMin

- Return the node containing the smallest element in the tree
- Start at the root and go left as long as there is a left child. The stopping point is the smallest element

```
Node*findMin(node* root)
{
   If(root==NULL)
      Return Null;
   Else if(root->left==Null)
      Return root
   Else
      Return(findMin(root->left)
}
```

findMax

- Finds the maximum element in BST
- Start at the root and go right as long as there is a right child. The stopping point is the largest element

```
Node*findMax(node*root)
{
   If(root==NULL)
      Return Null;
   Else if(root->right==Null)
      Return root
   Else
      Return(findMan(root->right)
}
```

delete

- When we delete a node, we need to consider how we take care of the children of the deleted node.
  - When a node is deleted the definition of a BST should be maintained.
- When a node is deleted four cases should be considered
  - Case1: Deleting a leaf node (a node with no child)
  - Case2: Deleting a node having only one child
  - Case3: Deleting a node having two child
  - Case4: Deleting a root node

delete

Three cases:

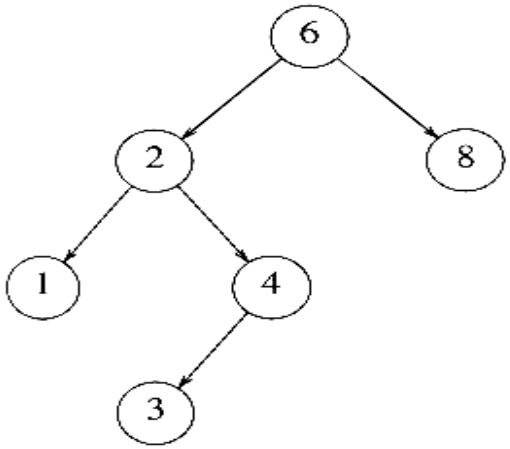
(1) the node is a leaf

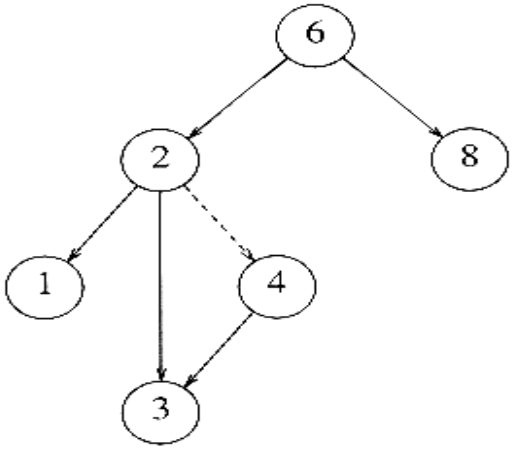
- Delete it immediately

(2) the node has one child

- Adjust a pointer from the parent to bypass that node

- Example delete node 4, make node 2 pointer point to node 3



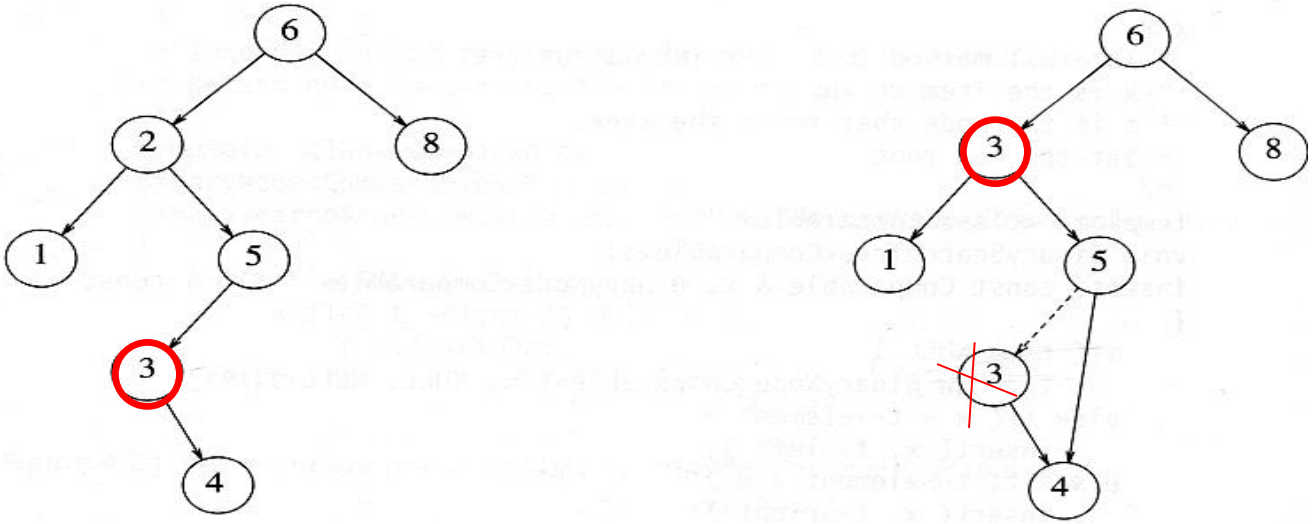


delete

(3) the node has 2 children

- Copy the node containing the largest element in the left(or the smallest element in the right)to the node to be deleted
- Delete the copied node

- The picture below shows deleting node2



- Time complexity = O(height of the tree)

Delete the root

- If BST has only one node, make root to point to nothing
  - Root=NULL
- Otherwise,
  - copy the node containing the largest element in the left(or the smallest element in the right)to the node to be deleted
  - Delete the copied node

Other operations BST might support

- getTreeSize()
- countLeafNodes()
- Equal()// given two binarySearchTree
- copyBinaryTree() //creates a new tree from existing one
- swapChild() // swaps left and right child

count number of nodes in a tree

```
int treeSize(node  *root)
{
        if(root == NULL)
                return 0;
        else
                return treeSize(root->left) + 1 + treeSize(root->right);
}
```

Counting leaf nodes on a binary tree

```
int countleaves(node root)
{
 int count;
 if(root==NULL)
  count=0;
 else if((root->left==NULL)&&(root->right==NULL))
   count=1;
 else
    count=countleaves(root->left) + countleaves(root->right);
 return count;
 }
```

Checking for equality of two binary trees

```
boolean equal(node root1,node root2)
 {
  if(root1==NULL&&root2==NULL)
   return true;
  else if(root1!=NULL&&root2!=NULL)
   if(root1->data==root2->data)
    return ((equal(root1->left,root2->left)) && (equal(root1->right,root2->right)));
  return false;
  }
```

Copy a binary tree

```
pnode copybt(node root)
{
 node p;
 if(root==NULL)
  return NULL;
 else
  {
    p=get_node();
    p->data=root->data;
    p->left=copybt(root->left);
    p->right=copybt(root->right);
    }
 return p;
}
```

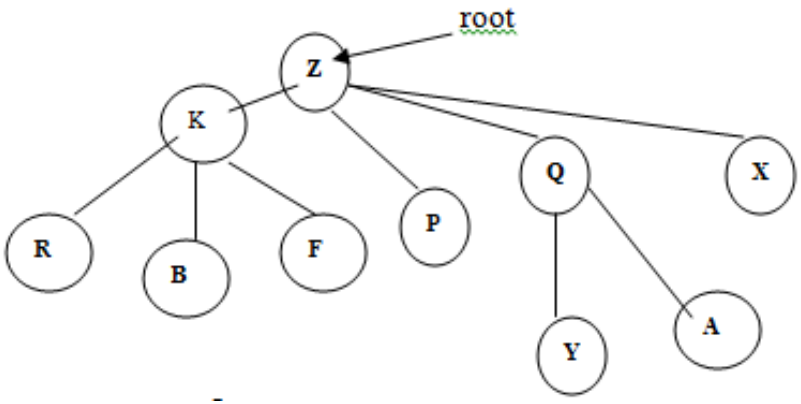
Swap the left and the right children of every node in a binary tree

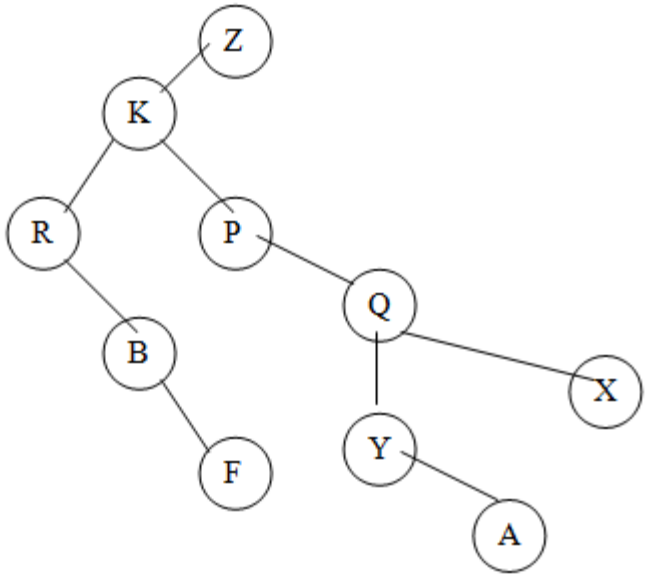
```
void swap(node root)
{
        node temp;
        if(root!=NULL)
         {
                temp=root->left;
                root->left=root->right;
                root->right=temp;
                swap(root->left);
                swap(root->right);
        }
}
```

General Tree

- A general tree is a tree in which a parent may have unrestricted number of children.
- General tree can be represented by binary tree
  - lptr of binary tree points to the left most child of a GT node
  - Rptr of binary tree points to the sibling node of a GT
- Accordingly, the GT shown in left can be represented by a binary tree to the right

Ex:





End of Chapter 6