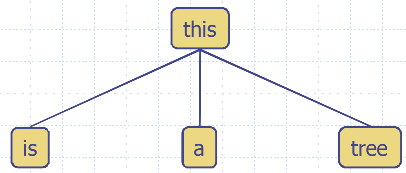
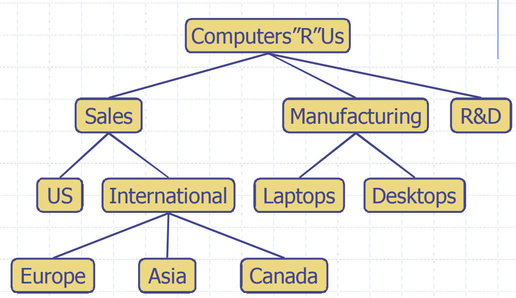
Trees



Dr. Md. Abul Kashem Mia, Professor, CSE Dept, BUET

What is a Tree

- A connected acyclic graph is a tree
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relationship
- Applications:
  - Organization charts
  - File systems
  - Programming environments

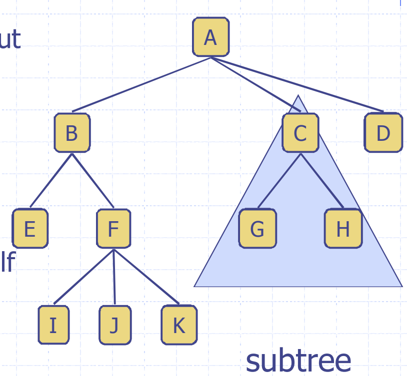


What is a Tree

- A connected acyclic graph is a tree.
- A tree *T* is a set of nodes in a parent-child relationship with the following properties:
  - *T* has a special node *r*, called the root of *T*, with no parent node
  - Each node *v* of *T*, different from *r*, has a unique parent node *u*
- A tree cannot be empty, since it must have at least one node – the root.

Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (Leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Descendants of a node: child, grandchild, grand-grandchild, etc.
- Depth of a node: number of ancestors, excluding the node itself
- Height of a tree: maximum depth of any node (3)
- Siblings: two nodes that are children of the same parent
- Subtree: tree consisting of a node and its descendants



Depth and Height

- The depth of a node *v* can be recursively defined as follows
  - If v is the root, then the depth of v is 0.
  - Otherwise, the depth of v is one plus the depth of the parent of v

Algorithm depth(T, v)
  if T.isRoot(v) **then**
    return 0
  else
    return 1 + depth(T, T.parent(v))

Running time: $O(1 + d_v)$, d_v is depth of v in T

In worst case $O(n)$, n is the number of nodes in T

Depth and Height

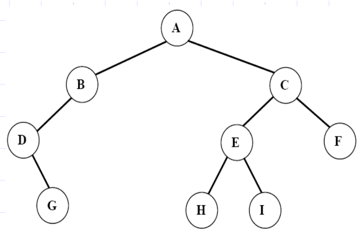
- The height of a node *v* can be recursively defined as follows
  - If v is a leaf node, then the height of v is 0.
  - Otherwise, the height of v is one plus the maximum height of a child of v

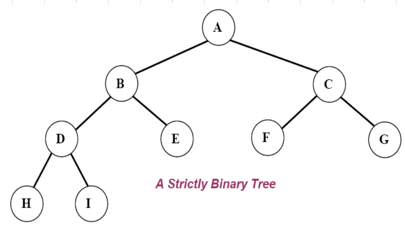
The height of a tree *T* is the height of the root of *T*

The height of a tree *T* is equal to the maximum depth of a leaf node of *T*

Ordered Trees

- A tree is ordered if there is a linear ordering defined for each child of each node.
- A binary tree is an ordered tree in which every node has at most two children.
- If each node of a tree has either zero or two children, the tree is called a proper (strictly) binary tree.





A Strictly Binary Tree

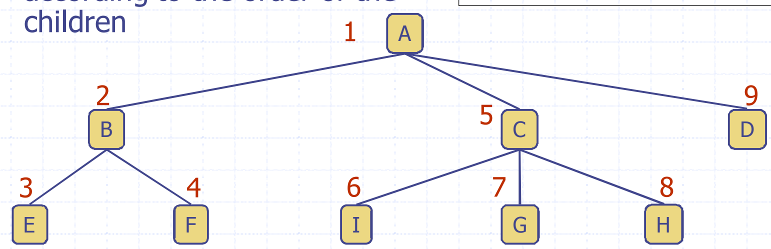
Traversal of Trees

- A traversal of a tree T is a systematic way of visiting all the nodes of T
- Traversing a tree involves visiting the root and traversing its subtrees
- There are the following traversal methods:
  - Preorder Traversal
  - Postorder Traversal
  - Inorder Traversal (of a binary tree)

Preorder Traversal

- In a preorder traversal, a node is visited before its descendants
- If a tree is ordered, then the subtrees are traversed according to the order of the children

Algorithm *preOrder*(*v*)
    visit(*v*)
    for each child *w* of *v*
        preorder (*w*)

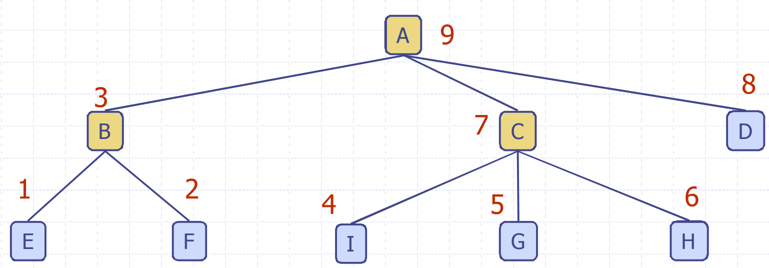


Preorder: *ABEFCIGHD*

Postorder Traversal

- In a postorder traversal, a node is visited after its descendants

Algorithm *postOrder*(*v*)
    for each child *w* of *v*
        postOrder (*w*)
    visit(*v*)

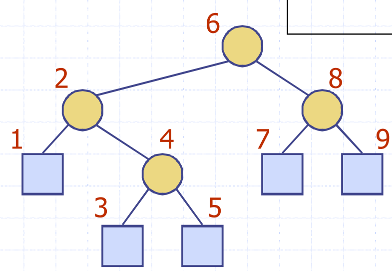


Postorder: *EFBIGHCDA*

Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree

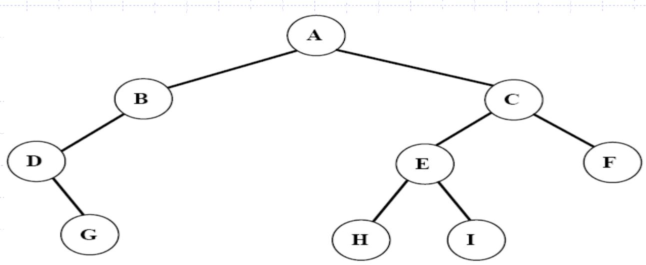
Algorithm *inOrder*(*v*)
    if *isInternal* (*v*)
        inOrder (*leftChild* (*v*))
    visit(*v*)
    if *isInternal* (*v*)
        inOrder (*rightChild* (*v*))



Inorder Traversal

Traversing a binary tree in *inorder*

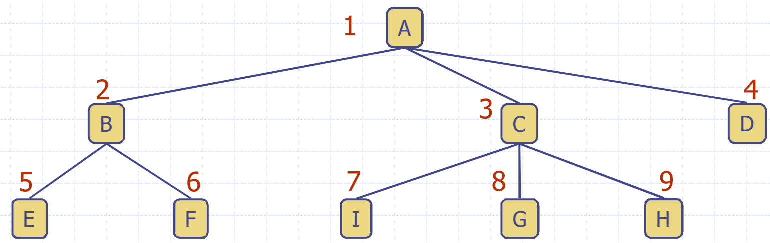
1. Traverse the ***left subtree*** in inorder.
2. Visit the ***root***.
3. Traverse the ***right subtree*** in inorder.



Inorder*: DGBAHEICF*

Level Order Traversal

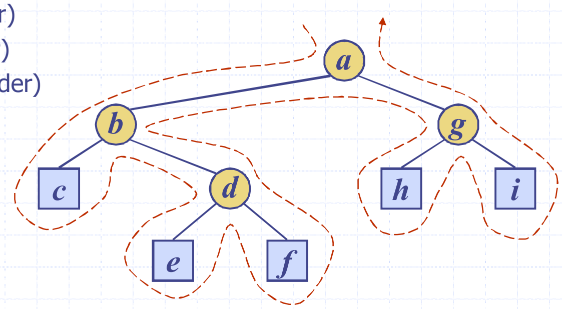
- In a level order traversal, every node on a level is visited before going to a lower level



Level order*: ABCDEFIGH*

Euler Tour Traversal

- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)



Preorder: ***a b c d e f g h i***

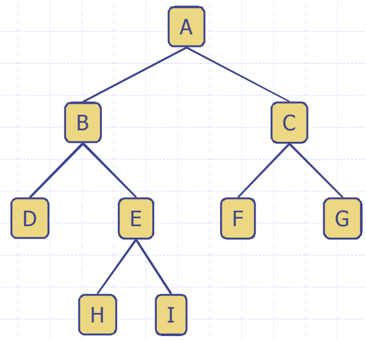
Inorder: ***c b e d f a h g i***

Postorder: ***c e f d b h i g a***

Euler Tour*: a b c b d e d f d b a g h g i g a*

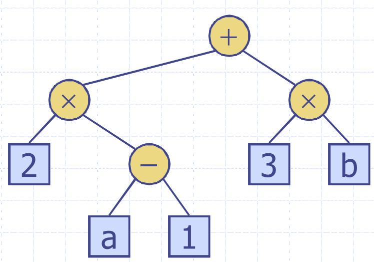
(Proper) Binary Tree

- A (proper) binary tree is a tree with the following properties:
  - Each internal node has two children
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a (proper) binary tree is either
  - a tree consisting of a single node, or
  - a tree whose root has an ordered pair of children, each of which is a proper binary tree
- Applications:
  - arithmetic expressions
  - decision processes
  - searching



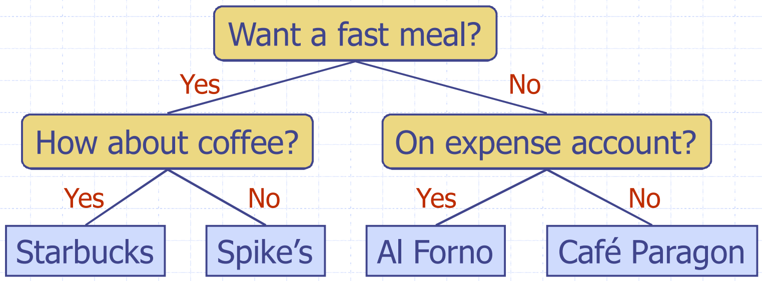
Arithmetic Expression Tree

- Binary tree associated with an arithmetic expression
  - internal nodes: operators
  - external nodes: operands
- Example: arithmetic expression tree for the expression $(2 \times (a - 1) + (3 \times b))$



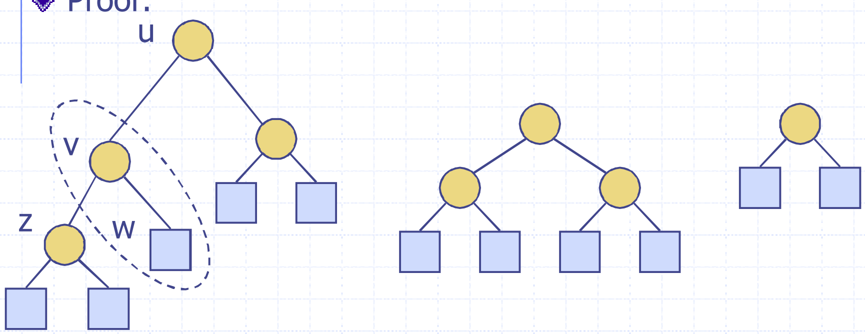
Decision Tree

- Binary tree associated with a decision process
  - internal nodes: questions with yes/no answer
  - external nodes: decisions
- Example: dining decision



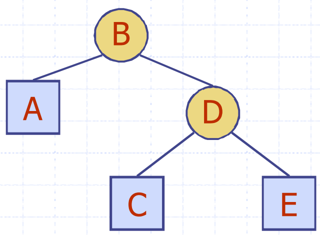
Properties of Binary Trees

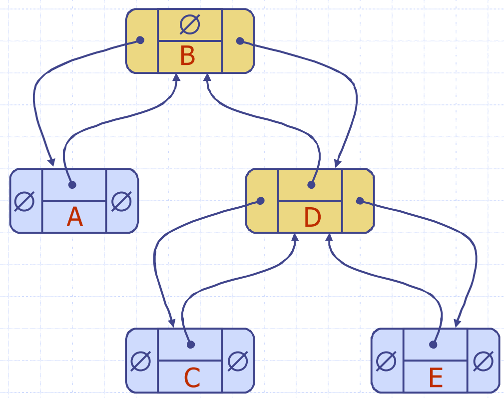
- In a (proper) binary tree T, the number of external nodes is 1 more than the number of internal nodes.
- Proof:



Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Left child node
  - Right child node





Codes for Creation of Binary Trees

- A binary tree can be created recursively. The program will work as follow:
  - Read a data in x.
  - Allocate memory for a new node and store the address in pointer p.
  - Store the data x in the node p.
  - Recursively create the left subtree of p and make it the left child of p.
  - Recursively create the right subtree of p and make it the right child of p.

Codes for Creation of Binary Trees

```
#include<stdio.h>
 typedef struct TreeNode {
    struct TreeNode *left;
    int data;
    struct TreeNode *right;
} TreeNode;
```


Codes for Creation of Binary Trees

```
TreeNode * createBinaryTree( ){
    TreeNode *p;
    int x;
    printf("Enter data(-1 for no data): ");
    scanf("%d", &x);
        if(x == -1)
        return NULL;
    p = (TreeNode*) malloc(sizeof(TreeNode));
    p->data = x;
    printf("Enter left child of %d: \n", x);
    p->left = createBinaryTree ( );
    printf("Enter right child of %d: \n",x);
    p->right = createBinaryTree ();
    return p;
}
```

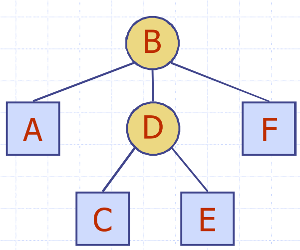
Codes for Creation of Binary Trees

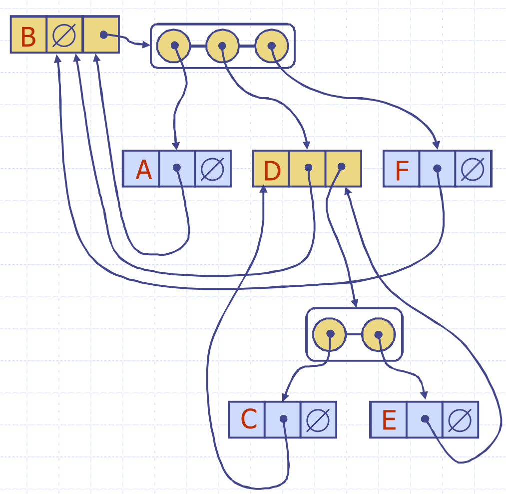
```
void preorder(TreeNode *t) {          //address of root node is passed in t
if(t != NULL) {
     printf("\n%d",  t->data);        //visit the root
     preorder(t->left);               //preorder traversal on left subtree
     preorder(t->right);              //preorder traversal on right subtree
  }
}

int main() {
    TreeNode *root;
    root = createBinaryTree ( );
    printf("\nThe preorder traversal of tree is: \n");
    preorder(root);
 return 0;
}
```


Linked Structure for General Trees

- A node is represented by an object storing
  - Element
  - Parent node
  - Children Container: Sequence of children nodes





Codes for Creation of General Trees

```
#include <stdio.h>
#include <conio.h>

typedef struct GTreeNode{
     int val;
     int NChild;
     struct GTreeNode *Child;
} GTreeNode;

void CreateGeneralTree(GTreeNode*, int);
void ShowGTNode(GTreeNode *R);
GTreeNode *Root = NULL;
```


Codes for Creation of General Trees

```
int main() {
    int i, val, n;          GTreeNode *NewNode ;
    printf("\nEnter Root Value: ");                   scanf("%d", &val);
    printf("Enter No. of Children of %d: ", val);     scanf("%d", &n);
    NewNode = new GTreeNode;
    if (n > 0)
       NewNode->Child = new GTreeNode[n];
    else
       NewNode->Child = NULL;
   NewNode->val=val;          NewNode->NChild = n;      GTreeNode empty = { 0 };
   for(i=0; i<n; i++)
      NewNode->Child[i] = empty;          //initially make them all Null
   Root = NewNode;                        // root points to newnode.
   CreateGeneralTree(Root, n);
   ShowGTNode(Root);
getch();   return 0;
}
```

Codes for Creation of General Trees

```
void CreateGeneralTree(GTreeNode *r, int n){
  int i, k, m;          char ch;
  for(i=0; i<n; i++){
    printf("\nEnter value for Child %d of %d: ", i+1, r->val);
    scanf("%d", &r->Child[i].val);
    r->Child[i].NChild = 0;    r->Child[i].Child = NULL;     }
  printf("\nDo You Wish to Enter Info of Child Nodes of %d? ", r->val);
  ch=getche();
  if(ch=='y' || ch=='Y'){
    for(k=0; k<n; k++){
      printf("\nEnter No. of Children of %d: ", r->Child[k].val);     scanf("%d", &m);
      r->Child[k].Nchild = m ;
      if (m > 0)
            r->Child[k].Child = new GTreeNode[m];
      else     r->Child[k].Child = NULL;
      CreateGeneralTree(&r->Child[k], m);   //Recursive
  } } }
```


Codes for Creation of General Trees

```
void ShowGTNode(GTreeNode *R) {
   int i, n;
   printf("\nInfo about %d:", R->val);
   n = R->NChild;
   printf("\tChildren: %d \t As ", n);
   for(i=0; i<n; i++)
        printf("%d", R->Child[i].val);
   for(i=0; i<n; i++)
      if(R->Child[i].NChild > 0)
          ShowGTNode(&(R->Child[i]));
}
```