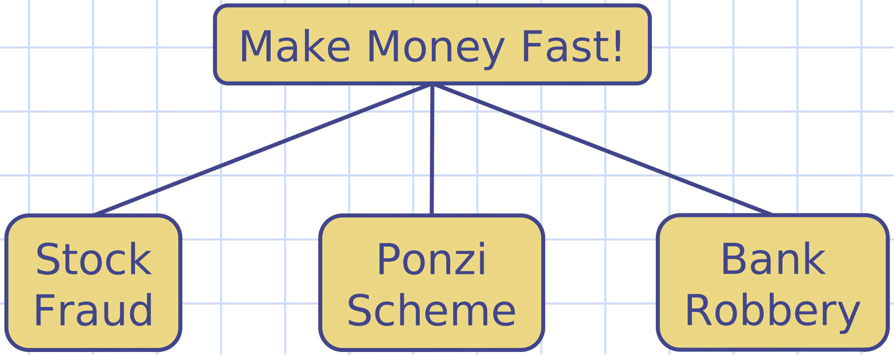
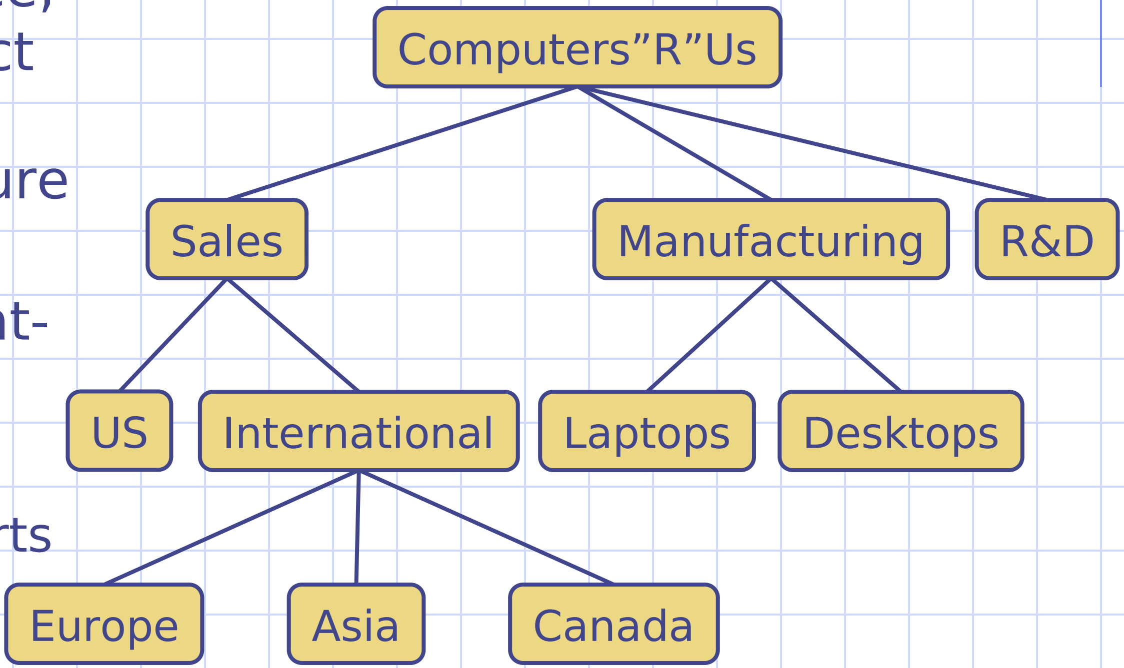
Trees



What is a Tree

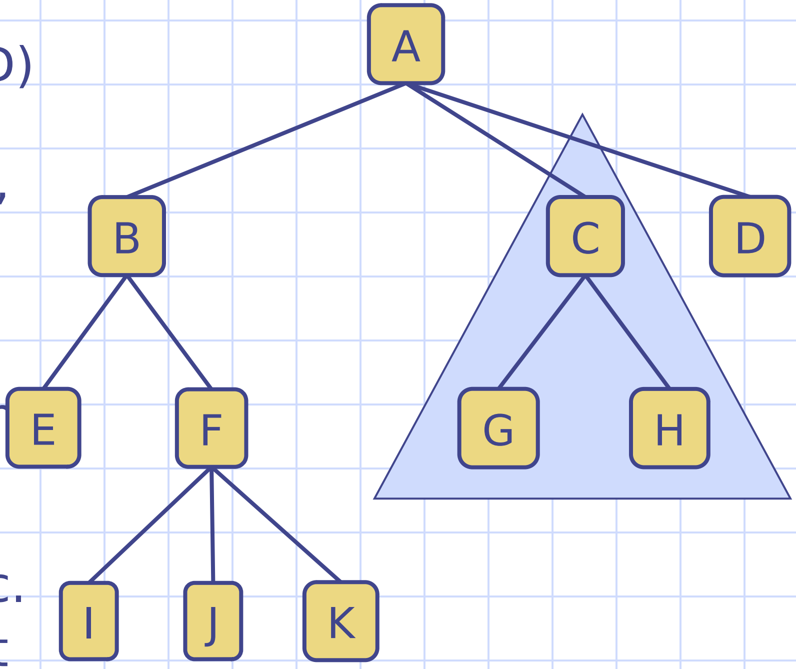
- In computer science, a tree is an abstract model of a hierarchical structure
- A tree consists of nodes with a parent-child relation
- Applications:
  - Organization charts
  - File systems
  - Programming environments



Tree Terminology

- Root: node without parent (A)
- Internal node: node with at least one child (A, B, C, F)
- External node (a.k.a. leaf): node without children (E, I, J, K, G, H, D)
- Ancestors of a node: parent, grandparent, grand-grandparent, etc.
- Depth of a node: number of ancestors
- Height of a tree: maximum depth of any node (3)
- Descendant of a node: child, grandchild, grand-grandchild, etc.
- Siblings: nodes with same parent

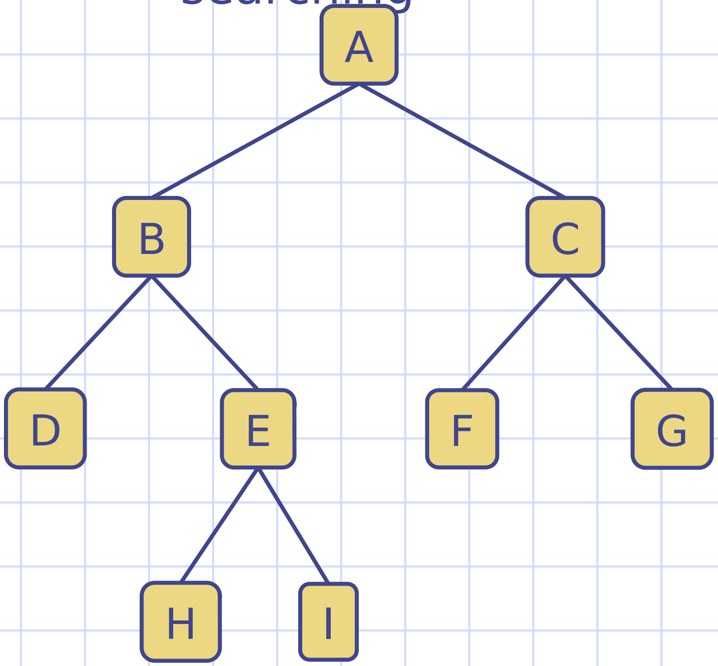
- Subtree: tree consisting of a node and its descendants



Binary Trees

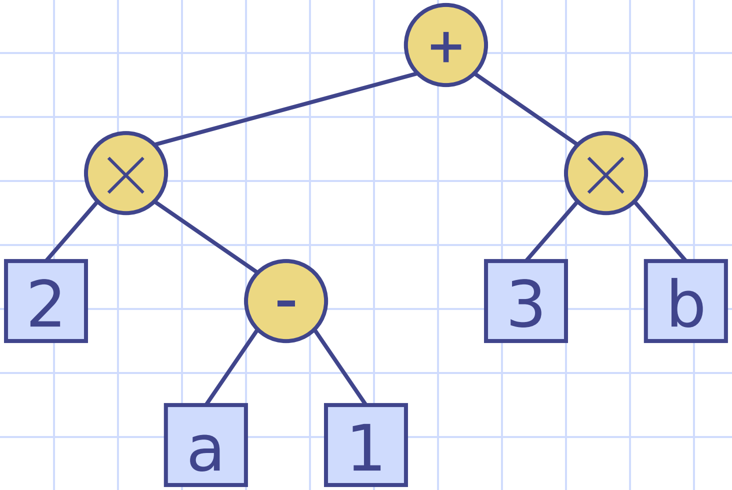
- A binary tree is a tree with the following properties:
  - Each internal node has at most two children (exactly two for proper binary trees)
  - The children of a node are an ordered pair
- We call the children of an internal node left child and right child
- Alternative recursive definition: a binary 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 binary tree

- Applications:
  - arithmetic expressions
  - decision processes
  - searching



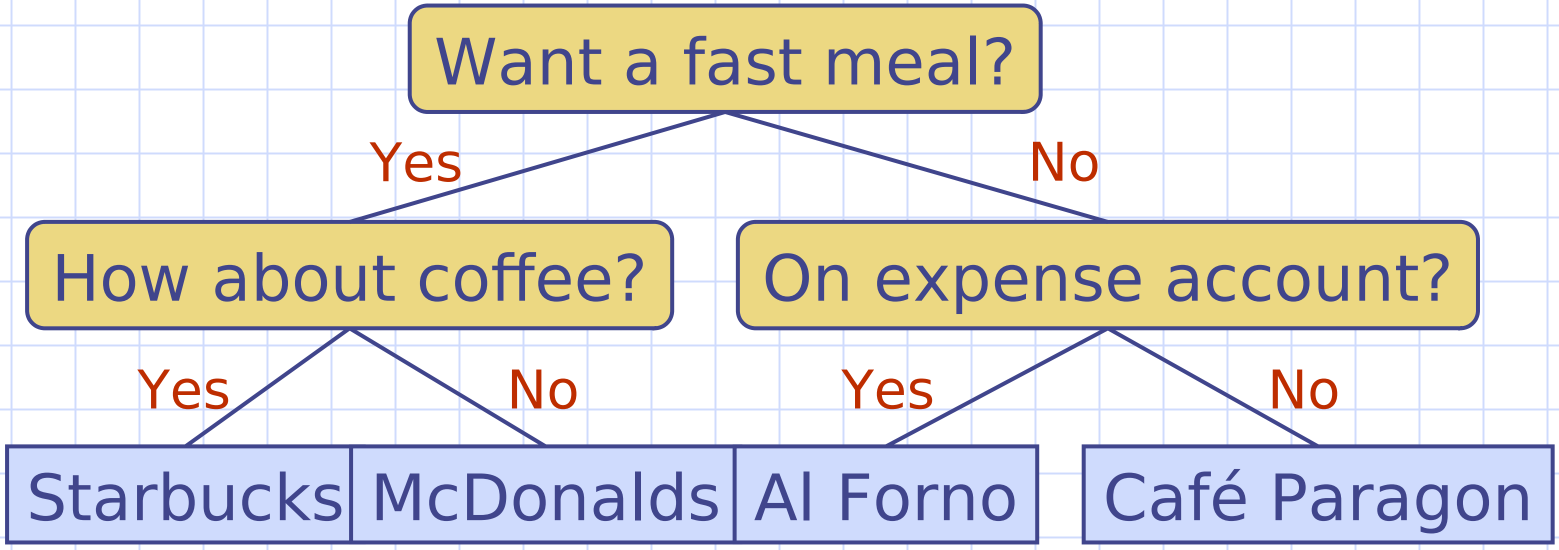
Arithmetic Expression Tree

- 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



Differences Between A Tree & A Binary Tree

- No node in a binary tree may have a degree more than 2, whereas there is no limit on the degree of a node in a tree.
- The subtrees of a binary tree are ordered; those of a tree are not ordered.

Differences Between A Tree & A Binary Tree

- The subtrees of a binary tree are ordered; those of a tree are not ordered

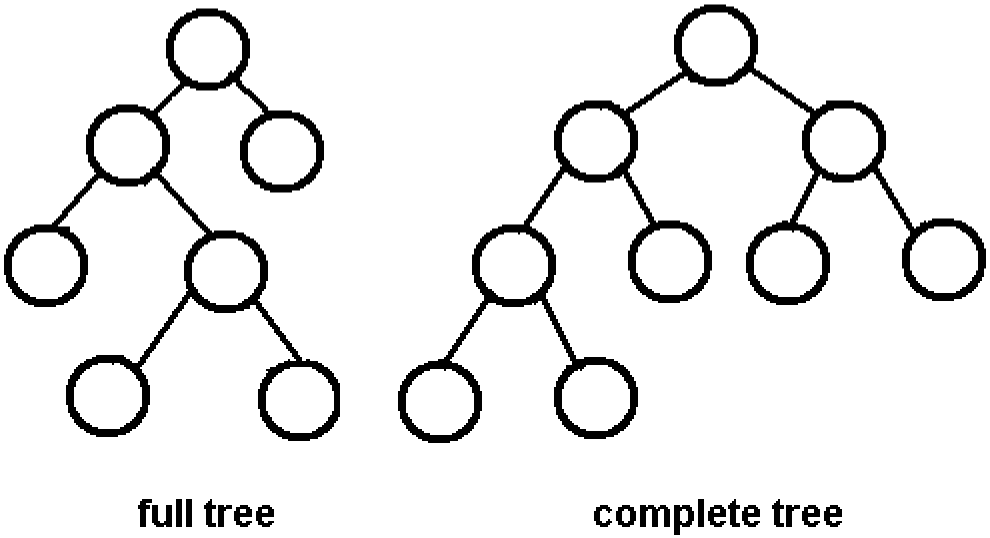
  - *Are different when viewed as binary trees*
  - *Are the same when viewed as trees*

Binary Trees

- **Full** binary tree :
  - All internal nodes have two children.
- **Complete** binary tree :
  - all levels except possibly the last are completely full,
  - the last level has all its nodes to the left side.
- **Perfect** binary Tree
  - A complete binary tree of height h has 2^h-1 internal nodes and 2^h leaves
  - Also: a binary tree with n nodes has height at least $\lfloor \lg n \rfloor$

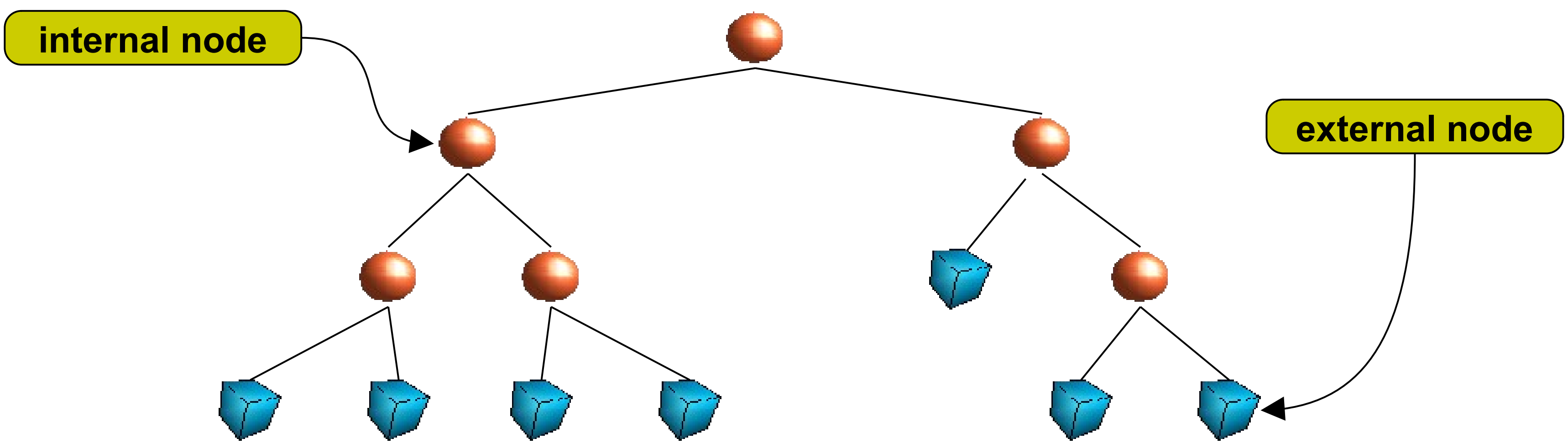
Binary Trees

- example



Properties Binary Trees

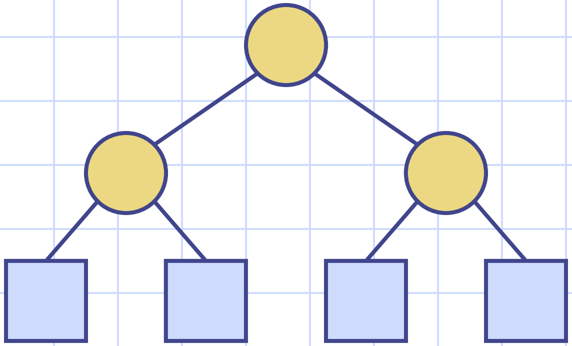
- We say that all internal nodes have at most two children
- External nodes have no children

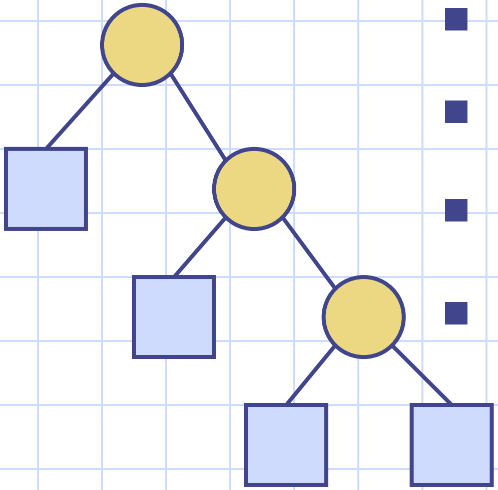


Properties of Proper (full) Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height

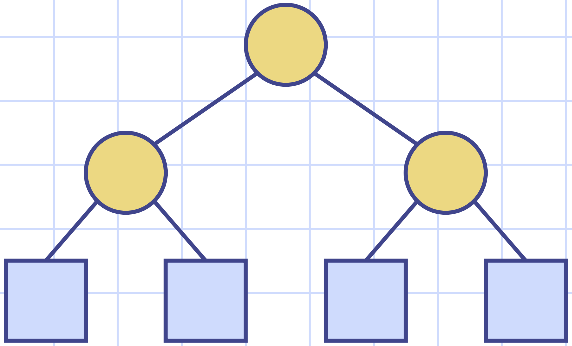
- Properties:
  - $e = i + 1$
  - $n = 2e - 1$
  - $h \leq i$
  - $h \leq (n - 1)/2$
  - $e \leq 2^h$
  - $h \geq \log_2 e$
  - $h \geq \log_2 (n + 1) - 1$





Properties of Proper Binary Trees

- Notation
  - ***n*** number of nodes
  - ***e*** number of external nodes
  - ***i*** number of internal nodes
  - ***h*** height



Property 1

Level d has at most 2^d nodes

0 = 1

1 = 2

$2 = 4 = 2^2$

$3 = 8 = 2 * 4 = 2^3$

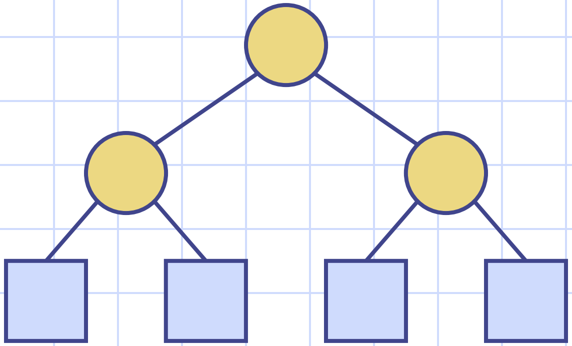
Let it be true for d-1 level

$d\text{-}1 = 2^{d-1}$

$D = 2 * 2^{d-1} = 2^d$

Properties of Proper full Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height



Property 2

A full binary tree of height h has (2^{h+1} – 1) nodes.

N = $2^0 + 2^1 + \dots 2^h$

= $2^{h+1} - 1$

Property 2.1

Height of a binary tree of N nodes = log(N)

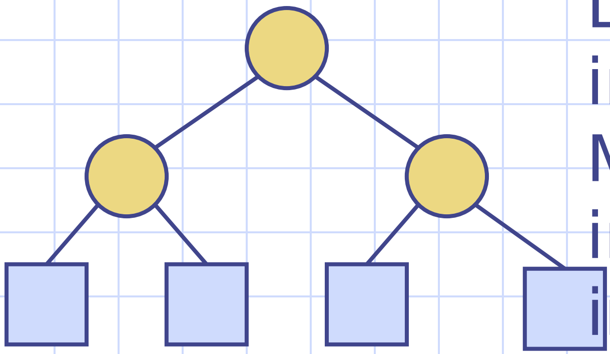
N = $2^{h+1} - 1$ = N + 1 = 2^{h+1}

h + 1 = log(N+1)

H = log(N+1) – 1 = Log(N)

Properties of Proper Binary Trees

- Notation
  - n number of nodes
  - e number of external nodes
  - i number of internal nodes
  - h height



Property 3

In a binary tree, the number of external nodes is 1 more than the number of internal nodes, i.e. $e = i + 1$.

Base – 0 internal node

Clearly true for one node.

Let it be true for N-1 internal node tree T

Make one external node internal to make it n internal node tree => 2 new external nodes, one old external node gone !!

Properties of Proper Binary Trees

- Notation
  - n number of nodes
  - e number of

Property 4

N = 2e – 1;

Proof ??? .. Based on previous ?

BinaryTree ADT

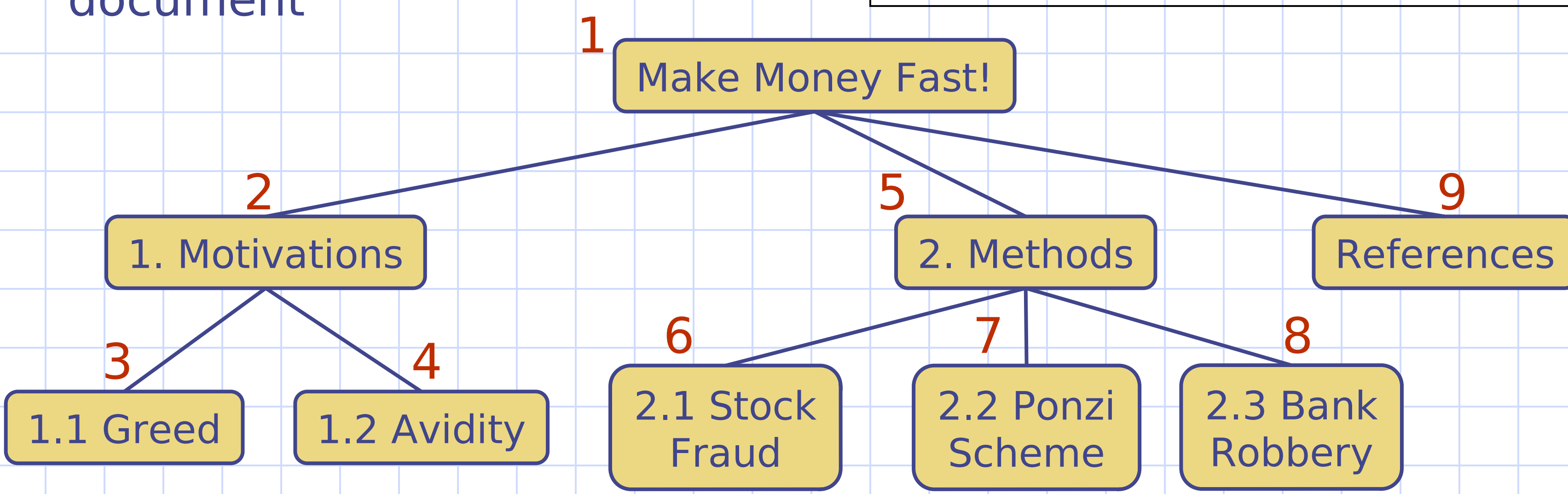
- The BinaryTree ADT extends the Tree ADT, i.e., it inherits all the methods of the Tree ADT
- Additional methods:
  - position p.left()
  - position p.right()
- Update methods may be defined by data structures implementing the BinaryTree ADT
- Proper binary tree: Each node has either 0 or 2 children

Preorder Traversal

- A traversal visits the nodes of a tree in a systematic manner
- In a preorder traversal, a node is visited before its descendants
- Application: print a structured document

Algorithm ***preOrder*(*v*)**

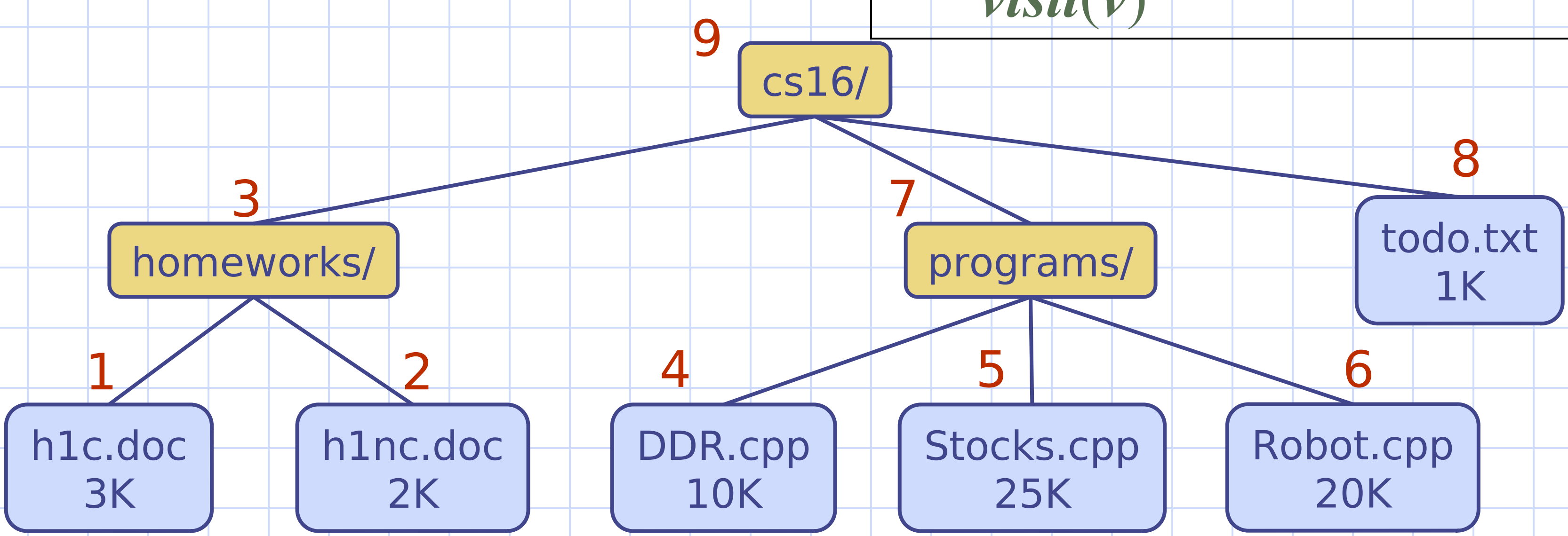
```
if ( v== null ) { return }
  visit(v)
        preorder (left)
    preorder(right)
```



Postorder Traversal

- In a postorder traversal, a node is visited after its descendants
- Application: compute space used by files in a directory and its subdirectories

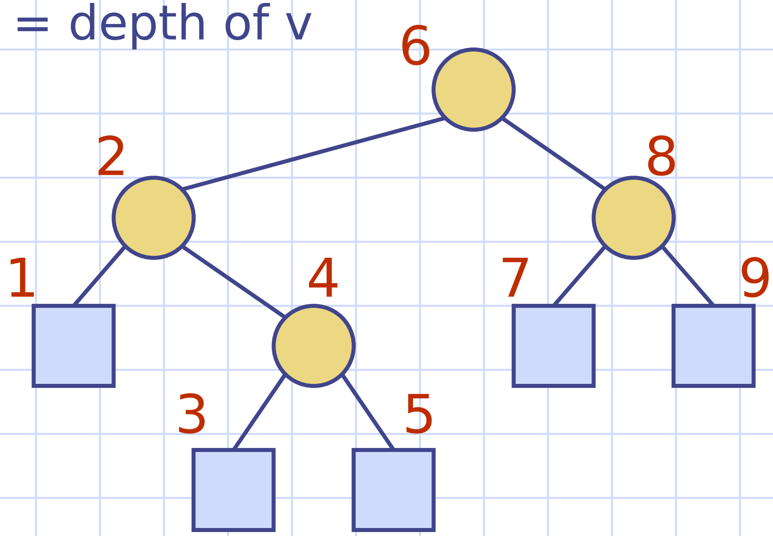
Algorithm *postOrder*(*v*)
    if (v==null) { return }
        postOrder (*left*)
        postOrder(right)
    visit(*v*)



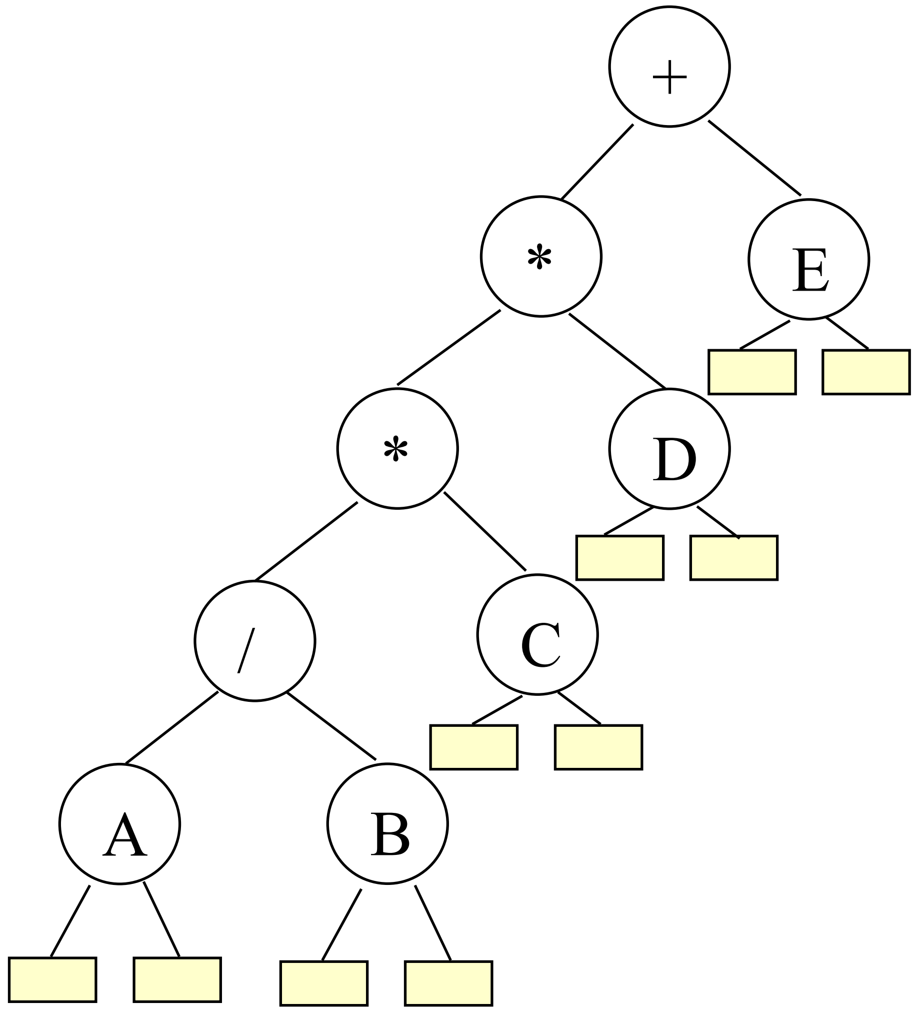
Inorder Traversal

- In an inorder traversal a node is visited after its left subtree and before its right subtree
- Application: draw a binary tree
  - x(v) = inorder rank of v
  - y(v) = depth of v

Algorithm ***inOrder***(***v***)
    if (v == null) {return}
    inOrder(v.left)
    visit(v)
    inOrder(v.***right***)



Arithmetic Expression Using BT



inorder traversal

A / B * C * D + E

infix expression

preorder traversal

+ * * / A B C D E

prefix expression

postorder traversal

A B / C * D * E +

postfix expression

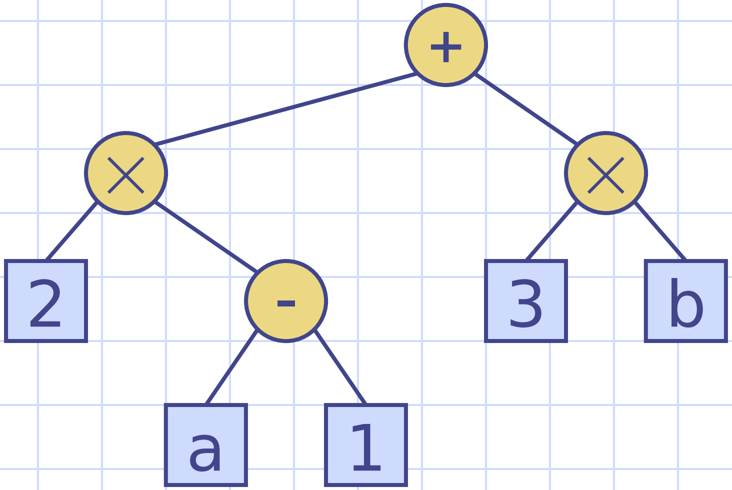
level order traversal

+ * E * D / C A B

Print Arithmetic Expressions

- Specialization of an inorder traversal
  - print operand or operator when visiting node
  - print “(“ before traversing left subtree
  - print “)“ after traversing right subtree

Algorithm *printExpression*(*v*)
    if ¬*v.isExternal*()
        print(“(’’)
        inOrder(*v.left*())
    print(*v.element*())
    if ¬*v.isExternal*()
        inOrder(*v.right*())
        print (“)’’)

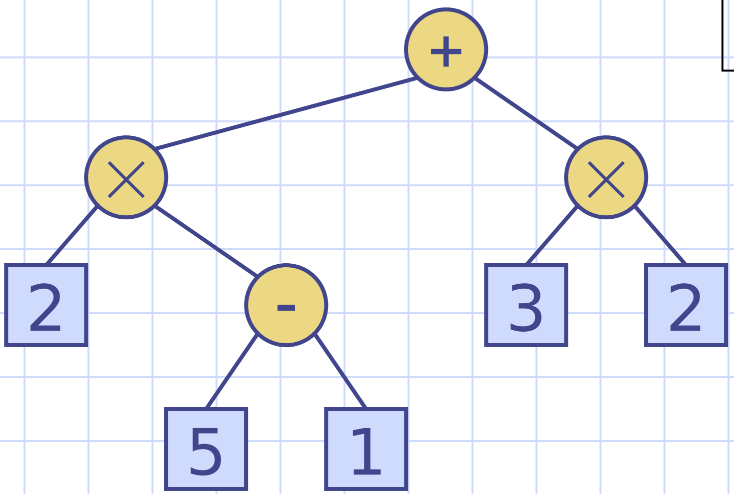


$((2 \times (a - 1)) + (3 \times b))$

Evaluate Arithmetic Expressions

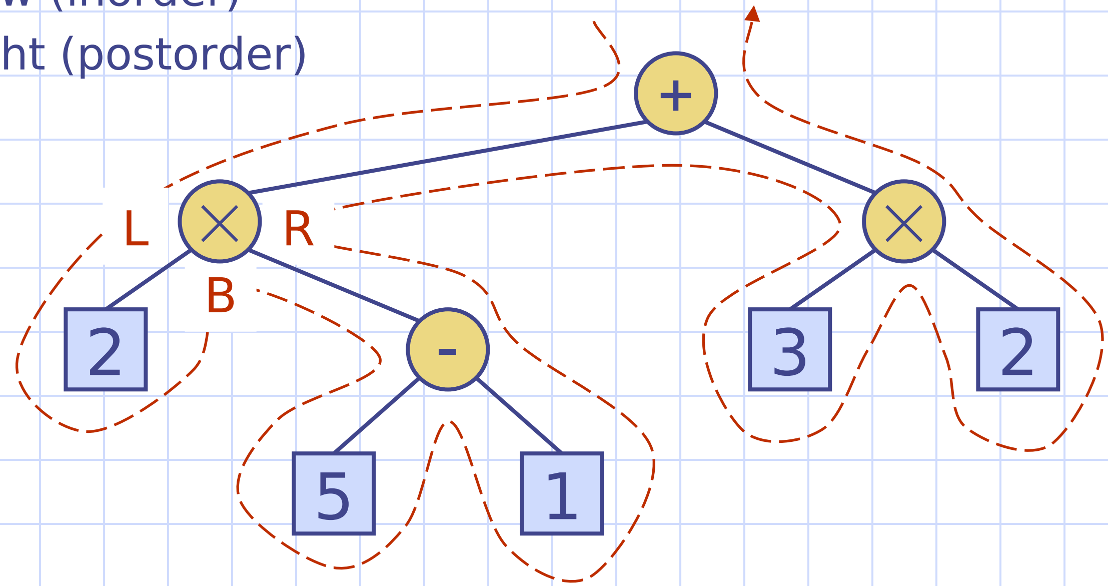
- Specialization of a postorder traversal
  - recursive method returning the value of a subtree
  - when visiting an internal node, combine the values of the subtrees

Algorithm *evalExpr*(v)
  if $v.isExternal()$
    return $v.element()$
  else
    $x \leftarrow evalExpr(v.left())$
    $y \leftarrow evalExpr(v.right())$
    $\Diamond \leftarrow$ operator stored at v
    return $x \Diamond y$



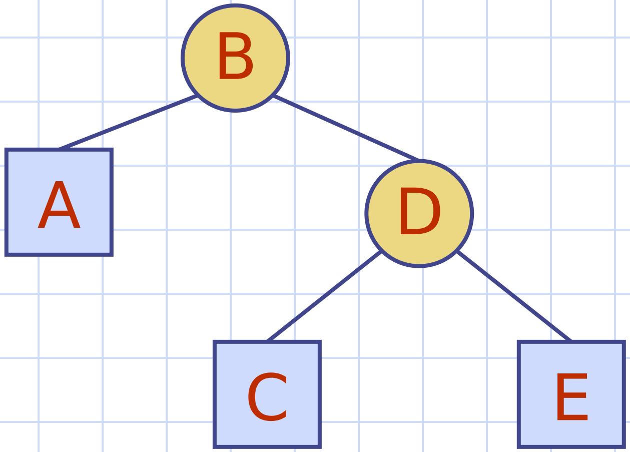
Euler Tour Traversal

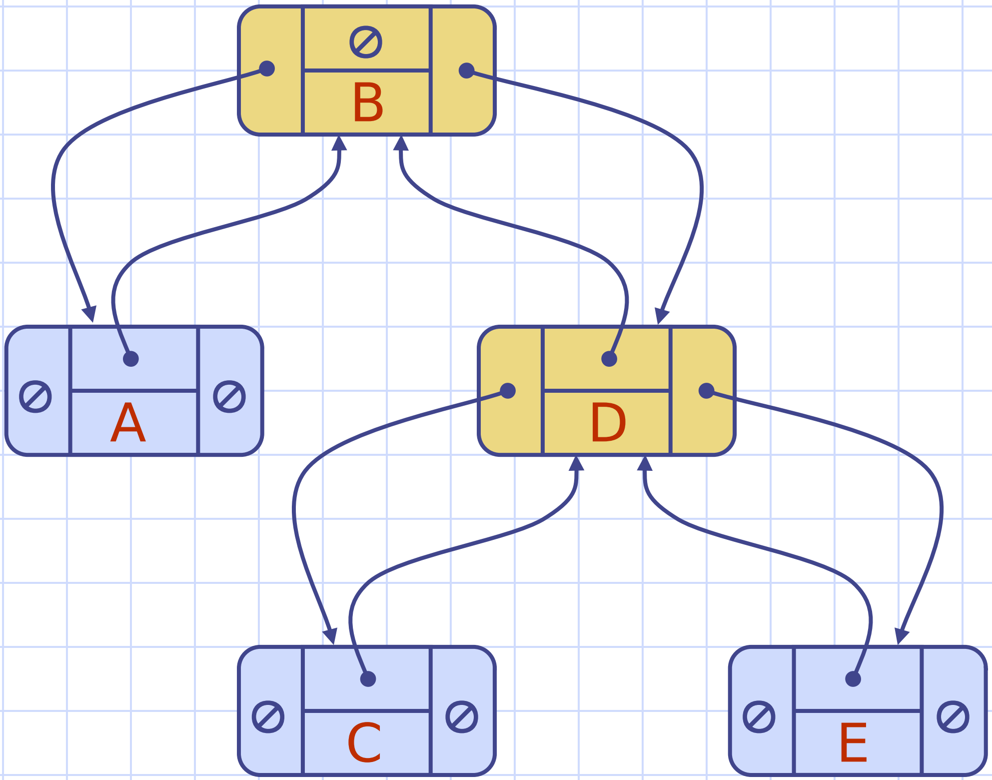
- Generic traversal of a binary tree
- Includes a special cases the preorder, postorder and inorder traversals
- Walk around the tree and visit each node three times:
  - on the left (preorder)
  - from below (inorder)
  - on the right (postorder)



Linked Structure for Binary Trees

- A node is represented by an object storing
  - Element
  - Left child node
  - Right child node
- Node objects implement the Position ADT





- Given a Binary tree, count the # of nodes in the tree.
- Main() {
-         n = count(root);
- }
- Int count(node) {
-      if (node == null) { return 0 }
-      return (1 + count(node.left) + count(node.right));
- }

- Given a Binary tree, count the # of leaf nodes in the tree.
- Main() {
-     n = lcount(root);
- }
- Int lcount(node) {
-         if (node == null) { return 0 };
-         if (node.left == null) && (node.right == null) { return 1};
-         return(lcount(node.left) + lcount(node.right));
- }

- }

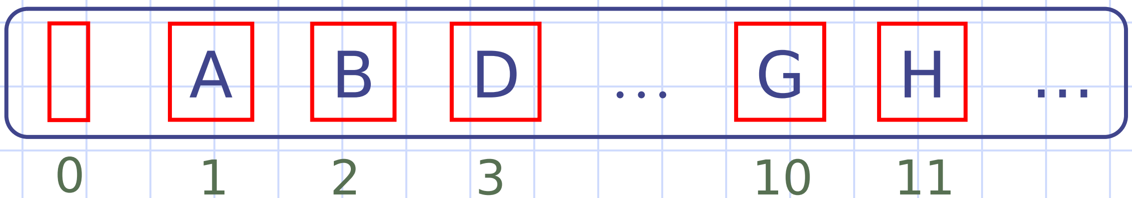
- Given a Binary tree, and a value, find the corresponding sibling node..

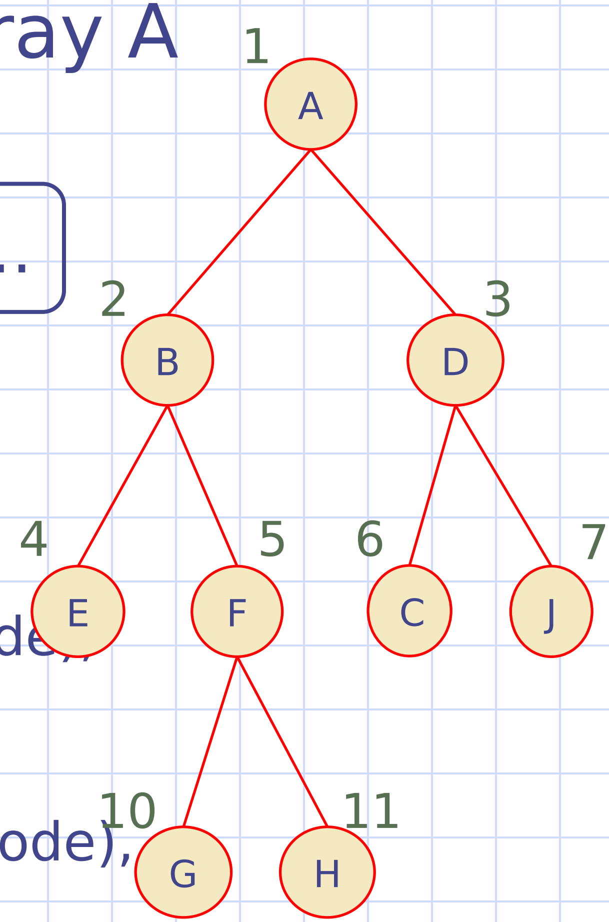
```
Main() {
      nd = sibling( root, 8 );
}
Node sibling( node, val ) {
    if ( node == null ) { return null }
    if ( node.val == val ) { return null }
    node tmp = null;
    if (node.left != null ) {
         if (node.left.val == val) { return (node.right) }
         tmp  =sibling(node.left, val);
    }
    if ( tmp == null && node.right != null ) {
         if ( node.right.val == val ) { return (node.left) }
         tmp = sibling(node.right, val);
   }
   return (tmp);


}
```

Array-Based Representation of Binary Trees

- Nodes are stored in an array A





- Node v is stored at A[rank(v)]
  - rank(root) = 1
  - if node is the left child of parent(node), rank(node) = 2 · rank(parent(node))
  - if node is the right child of parent(node), rank(node) = 2 · rank(parent(node)) + 1

A Binary-tree Node Class

```
Class Node {
    int data;
    Node  left;
    Node  right;
    void Node () { };
};
class BinTree {
   public:
    TreeNode( ) { root = NULL; };
    void insert ( int key );
    Node *rinsert( Node *s, int  key);
    void inTraverse( Node *s );
    void preTraverse( Node *s );
    void postTraverse( Node *s );
};
```

- Given a binary tree, count the number of nodes.
- NODE {
- int val;
- NODE lchild;
- Node rchild; }
- Main() {
- n = count(root) {
- }
- Count (Node tmp) { int n = 0;
- if (tmp == null) { return 0; }
- else return 1 + count(tmp.lchild) + count(tmp.rchild)

- Given a binary tree, count the number of leaf nodes.
- NODE {
- int val;
- NODE lchild;
- Node rchild; }
- Main() {
- n = lcount(root) {
- }
- lCount (Node tmp) {
- if (tmp == null) { return 0 }
- else if (tmp.lchild == null) and (tmp.rchild == null) { return 1 }
- else return lcount(tmp.lchild) + lcount(tmp.rchild) }

- Given a binary tree, and a node N, return the sibling of N. If none, return NULL.
  NODE { int val;
- NODE lchild;
- Node rchild; }
- Main() {
- nd = sib(root, N) {
- }
- Node sib(Node tmp, NODE n) {
- if (tmp == null) { return NULL }
- else if (tmp == n) { return null }
- else if (tmp.lchild == n) return { tmp.rchild }
- else if (tmp.rchild == n) return { tmp.lchild }
- t = sib(tmp.lchild, n) if (t == null) { t = sib(tmp.rchild, n) } return t;

- Given a binary tree, and a node N, return the parent of N. If none, return NULL. NODE { int val;
- NODE lchild;
- Node rchild; }
- Main() {
- nd = par(root, N) {
- }
- Node par(Node tmp, NODE n) {
- if (tmp == null) { return null };
- if (tmp.lchild == n) | (tmp.rchild == n) { return tmp }
- l = par(tmp.lchild,n); if (l == null) return par(tmp.rchild, n);
- else return l;

- Given a binary tree, find the height of the tree
- NODE {   int val;
-               NODE lchild;
-               Node rchild; }
- Main() {
-     h = ht(root) {   }
- int ht(Node tmp) {   if (tmp == null) return -1;
- Return 1 + max (ht(tmp.lchild), ht (tmp.rchild)

- Given a binary tree having integer values,there is at least one node. find the max value;
- NODE { int val;
- NODE lchild;
- Node rchild; }
- Main() {
- h = max(root) { }
- int max(Node tmp){
- if(tmp.lchild == null)&&(tmp.rchild == null)ret tmp.val;
- If (tmp.lchild == null) && (tmp.rchild != null) return gt(tmp.val, max(tmp.rchild)
- If (tmp.lchild != null) && (tmp.rchild == null) return gt (tmp.val, max(tmp.lchild);
- Else return greater(max(tmp.val), tmp.lchild), max(tmp.rchild)

Binary Tree Class

- class Tree {
- public:
- typedef int datatype;
- Tree(TreeNode *rootPtr=NULL){this->rootPtr=rootPtr;};
- TreeNode *search(datatype x);
- bool insert(datatype x);   TreeNode * remove(datatype x);
- TreeNode *getRoot(){return rootPtr;};
- Tree *getLeftSubtree();   Tree *getRightSubtree();
- bool isEmpty(){return rootPtr == NULL;};
- private:
- TreeNode *rootPtr;
- };