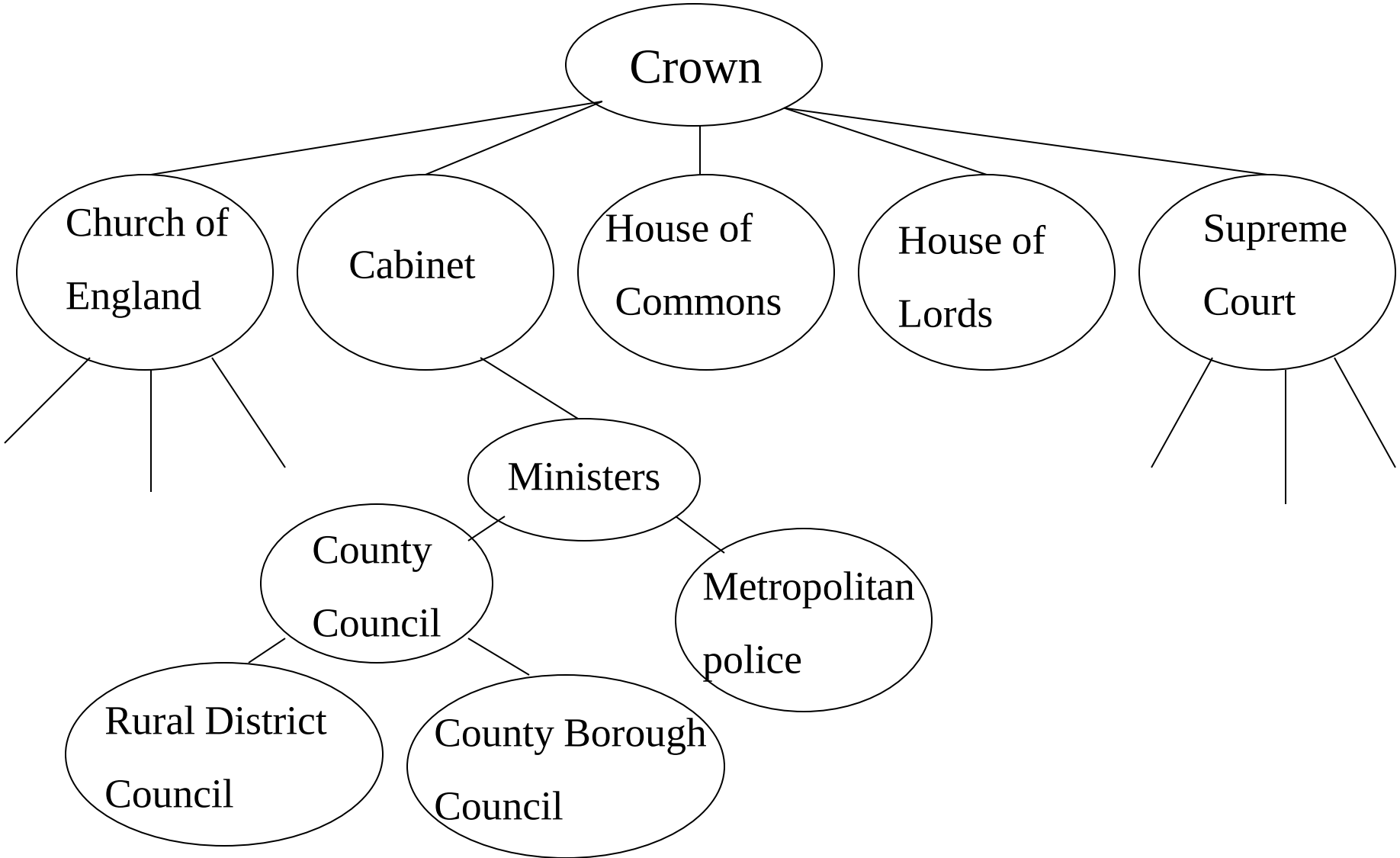
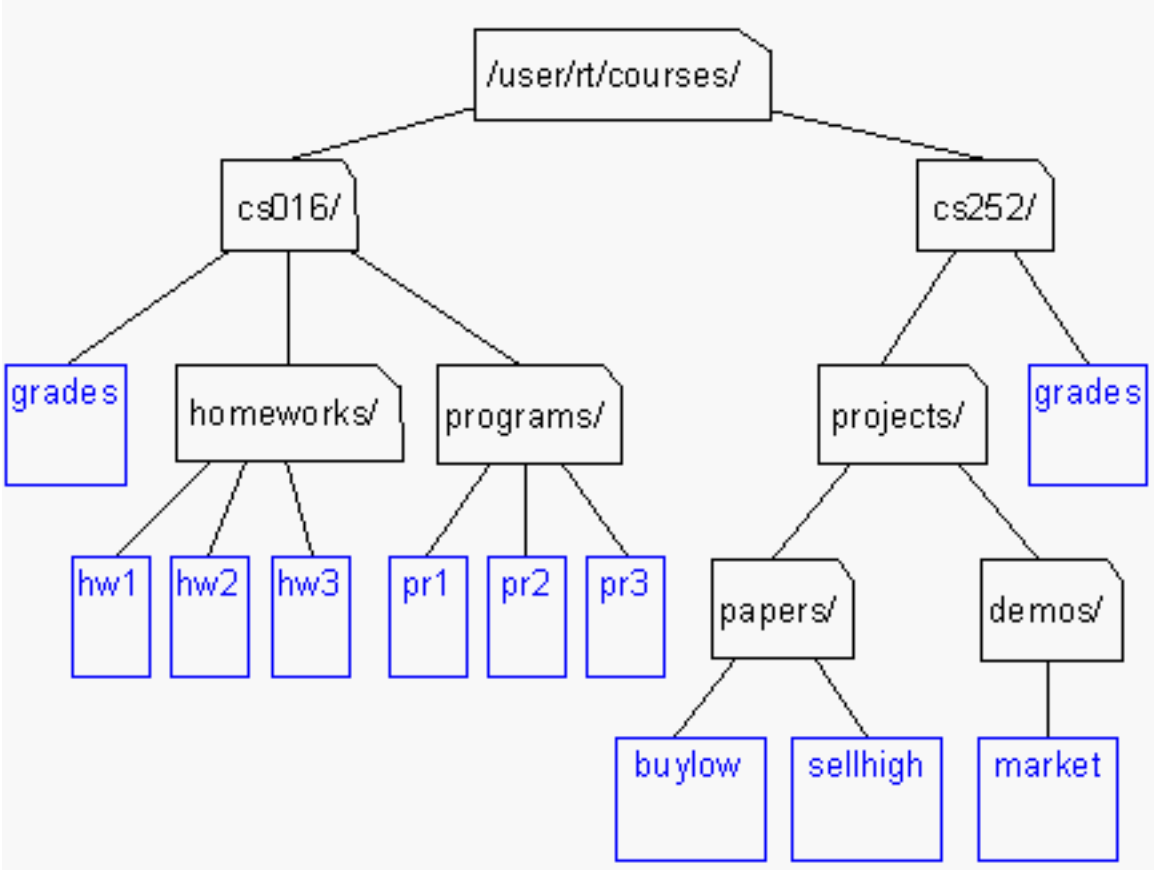
# Trees. Binary Trees.

# The British Constitution



# More Trees Examples

- Unix / Windows file structure

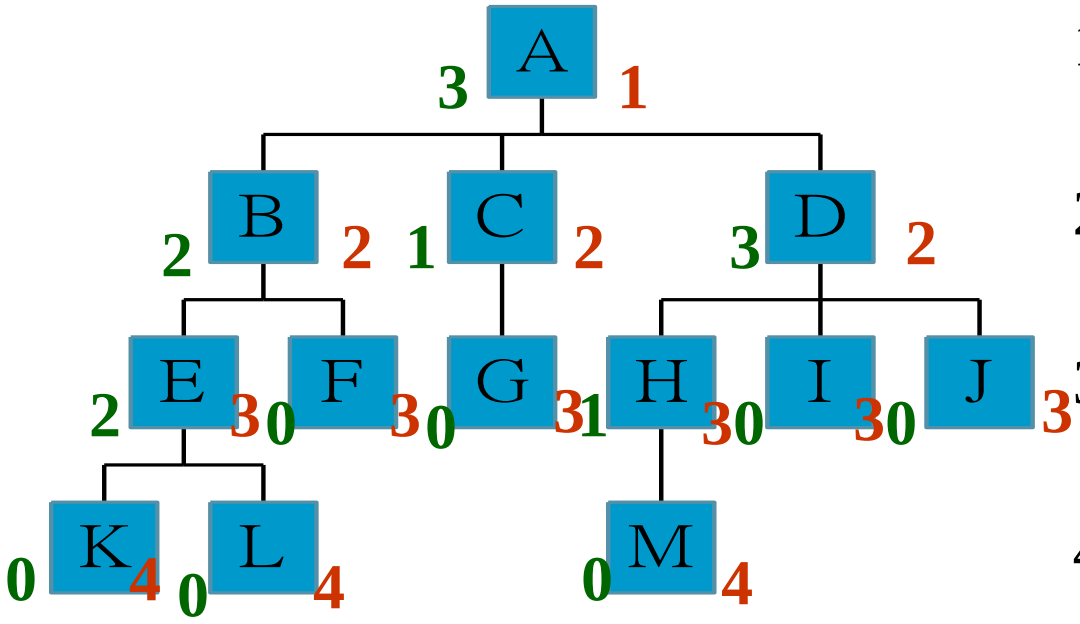


# *Definition of Tree*

- A tree is a finite set of one or more nodes such that:
- There is a specially designated node called the root.
- The remaining nodes are partitioned into n>=0 disjoint sets $T_1$, ..., $T_n$, where each of these sets is a tree.
- We call $T_1$, ..., $T_n$ the subtrees of the root.

# *Level and Depth*

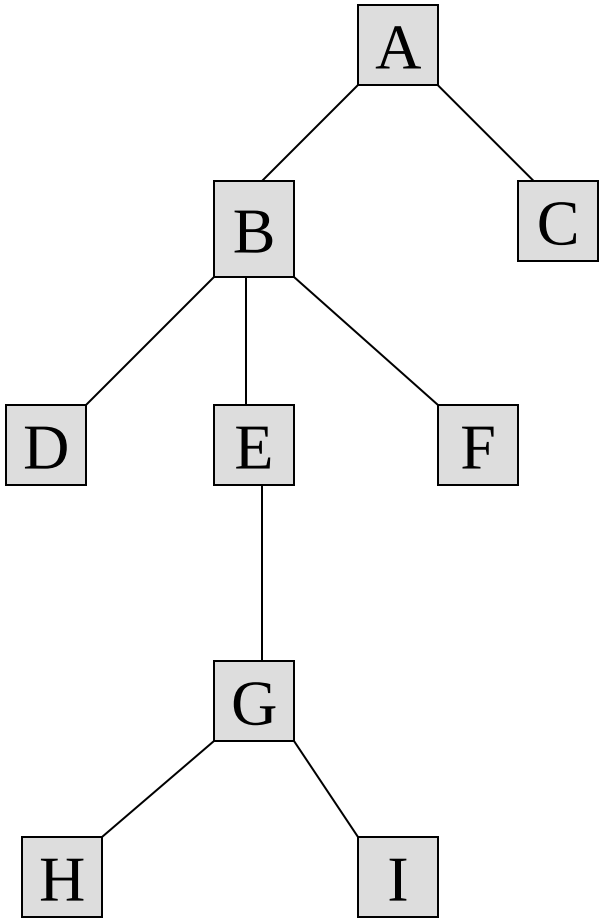
node (13)
degree of a node
leaf (terminal)
nonterminal
parent
children
sibling
degree of a tree (3)
ancestor
level of a node
height of a tree (4)



# *Terminology*

- The degree of a node is the number of subtrees of the node
  - The degree of A is 3; the degree of C is 1.
- The node with degree 0 is a leaf or terminal node.
- A node that has subtrees is the *parent* of the roots of the subtrees.
- The roots of these subtrees are the *children* of the node.
- Children of the same parent are *siblings*.
- The ancestors  of a node are all the nodes along the path from the root to the node.

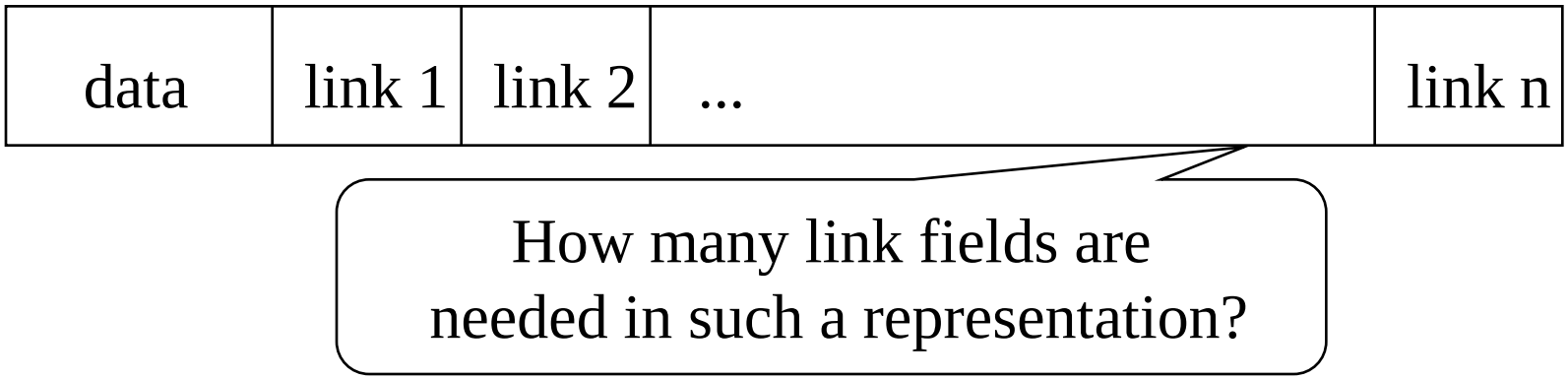
# Tree Properties




| Property | Value |
| --- | --- |
| Number of nodes | |
| Height | |
| Root Node | |
| Leaves | |
| Interior nodes | |
| Number of levels | |
| Ancestors of H | |
| Descendants of B | |
| Siblings of E | |
| Right subtree | |

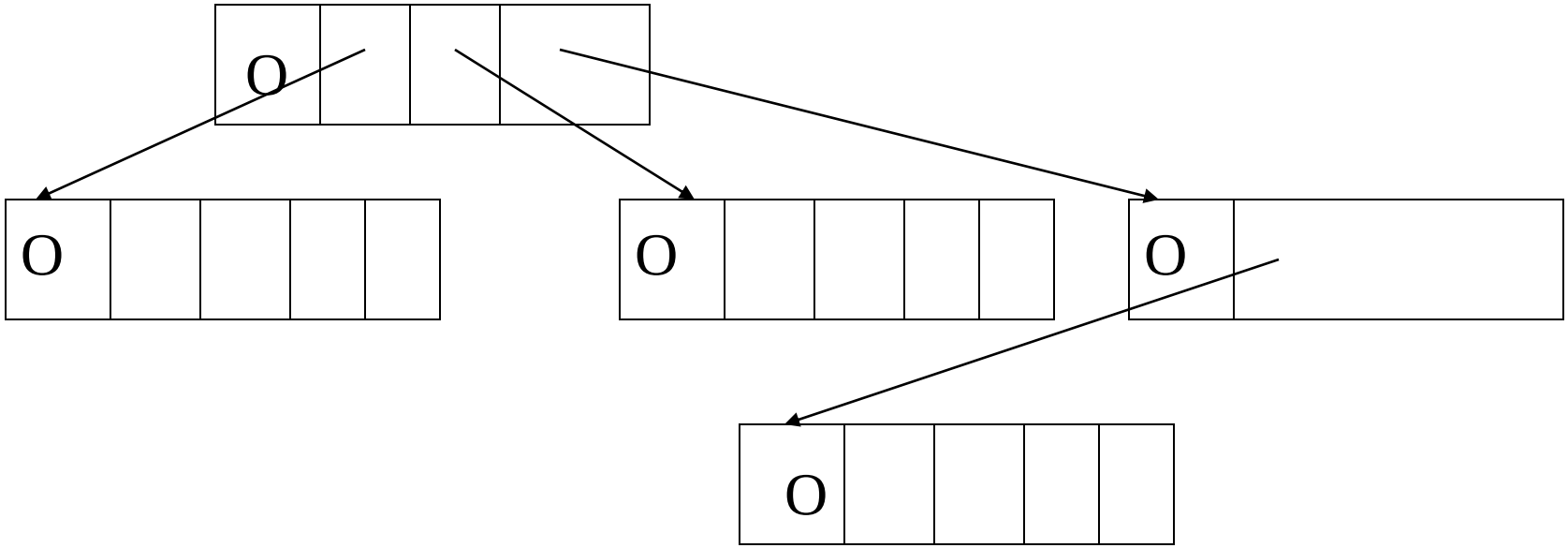
# Representation of Trees

- List Representation
  - ( A ( B ( E ( K, L ), F ), C ( G ), D ( H ( M ), I, J ) ) )
  - The root comes first, followed by a list of sub-trees

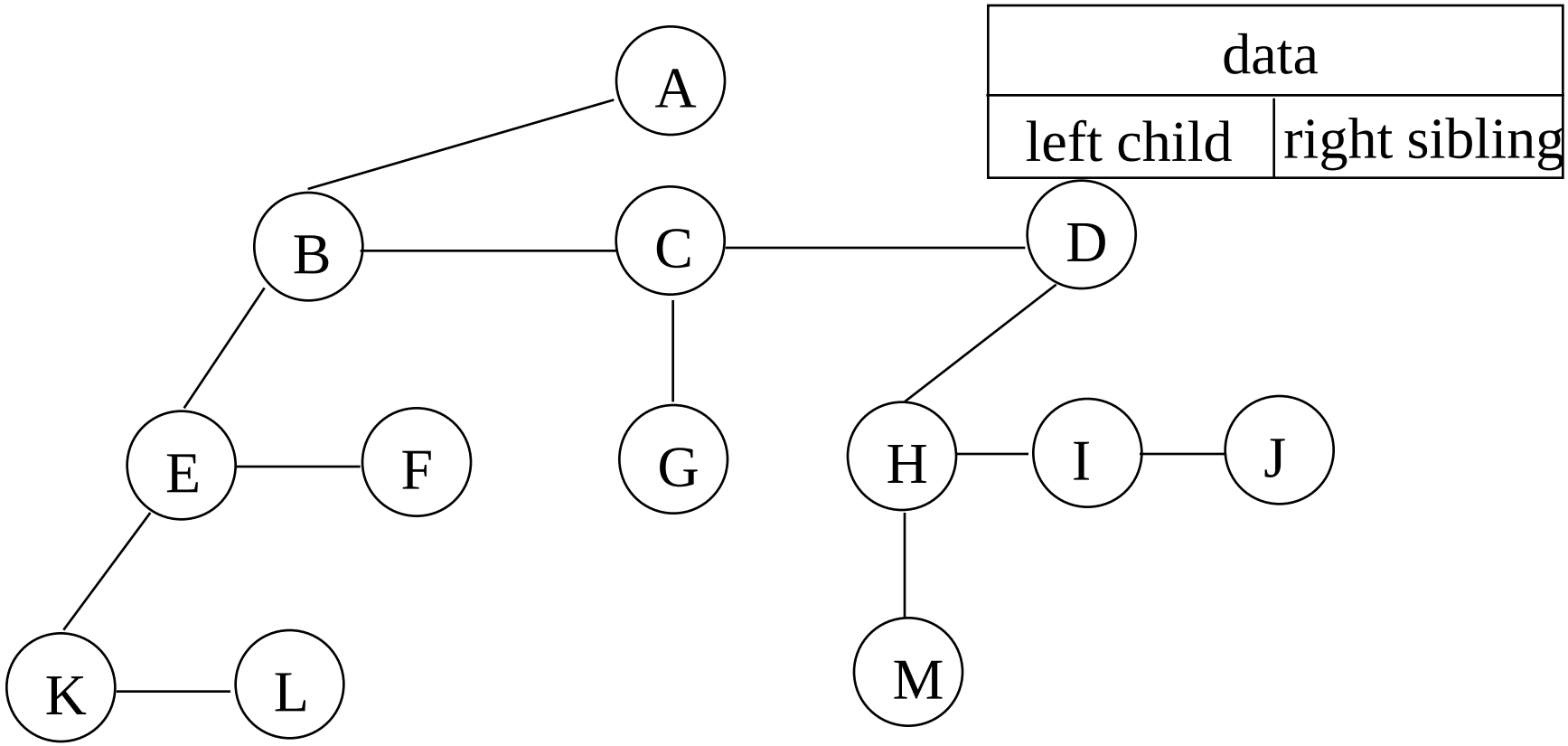


# A Tree Node

- Every tree node:
  - object – useful information
  - children – pointers to its children nodes



# Left Child - Right Sibling



# TREE ADT

- Objects: any type of objects can be stored in a tree
- Methods:
- accessor methods
  - root() – return the root of the tree
  - parent(p) – return the parent of a node
  - children(p) – returns the children of a node
- query methods
  - size() – returns the number of nodes in the tree
  - isEmpty() - returns true if the tree is empty
  - elements() – returns all elements
  - isRoot(p), isInternal(p), isExternal(p)

# Tree Implementation

```
typedef struct tnode {
    int key;
    struct tnode* lchild;
    struct tnode* sibling;
} *ptnode;
```

- Create a tree with three nodes (one root & two children)
- Insert a new node (in tree with root R, as a new child at level L)
- Delete a node (in tree with root R, the first child at level L)

# Tree Traversal

- Two main methods:
  - Preorder
  - Postorder
- Recursive definition

- PREorder:
  - visit the root
  - traverse in preorder the children (subtrees)
- POSTorder
  - traverse in postorder the children (subtrees)
  - visit the root

# PREORDER

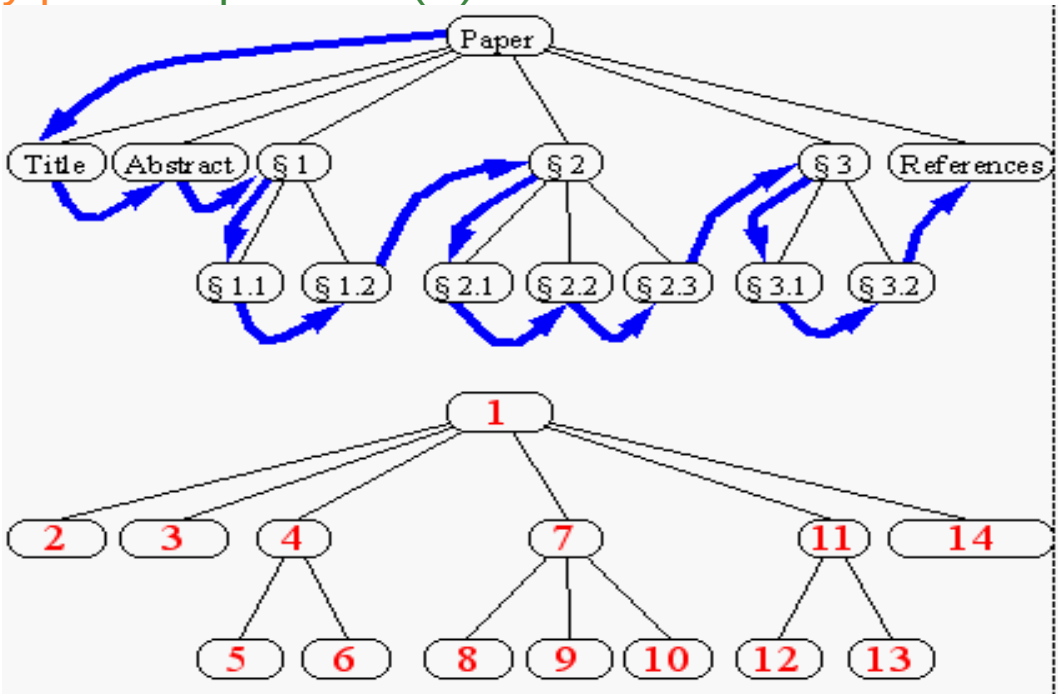
- **preorder traversal**

  **Algorithm** preOrder(v)

  “visit” node v

  **for each** child w of v **do**

  recursively perform preOrder(w)



# POSTORDER

- **postorder traversal**

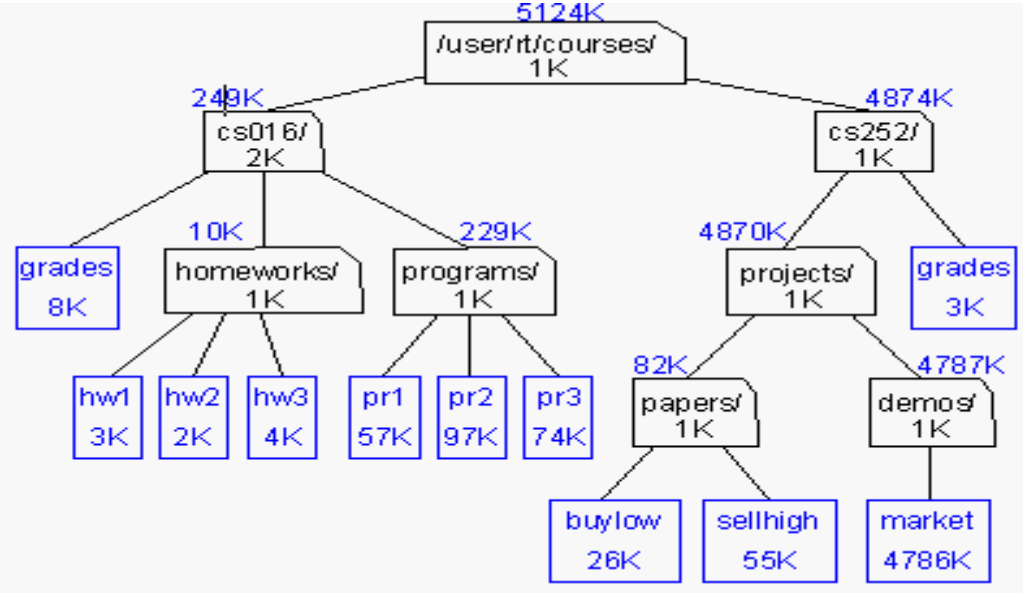
  **Algorithm** postOrder(v)

    **for each** child w of v **do**

      recursively perform postOrder(w)

    "visit" node v

- du (**disk usage) command in Unix**



# Preorder Implementation

```
public void preorder(ptnode t) {
   ptnode ptr;
   display(t->key);
   for(ptr = t->lchild; NULL != ptr; ptr = ptr->sibling) {
         preorder(ptr);
    }
}
```

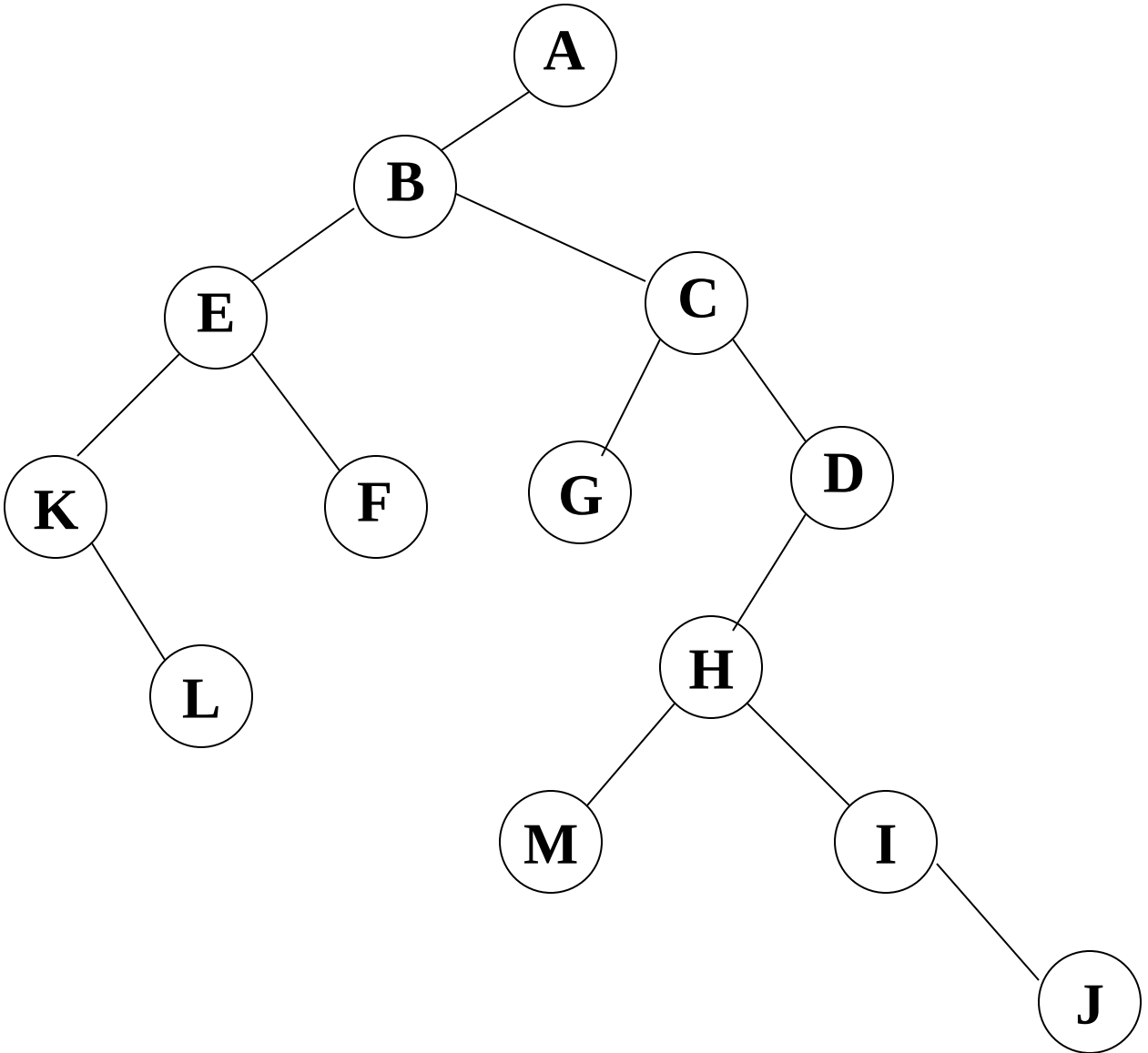
# Postorder Implementation

```
public void postorder(ptnode t) {
   ptnode ptr;
   for(ptr = t->lchild; NULL != ptr; ptr = ptr->sibling) {
         postorder(ptr);
    }
   display(t->key);
}
```

# *Binary Trees*

- A special class of trees: max degree for each node is 2
- Recursive definition: A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called *the left subtree* and *the right subtree*.
- Any tree can be transformed into binary tree.
  - by left child-right sibling representation

# EXAMPLE



# *ADT Binary Tree*

objects: a finite set of nodes either empty or consisting of a root node, left *BinaryTree*, and right *BinaryTree*.

method:

for all *bt*, *bt1*, *bt2* ∈ *BinTree*, *item* ∈ *element*

*Bintree* create()::= creates an empty binary tree

*Boolean* isEmpty(*bt*)::= if (*bt*==empty binary tree) return *TRUE* else return *FALSE*

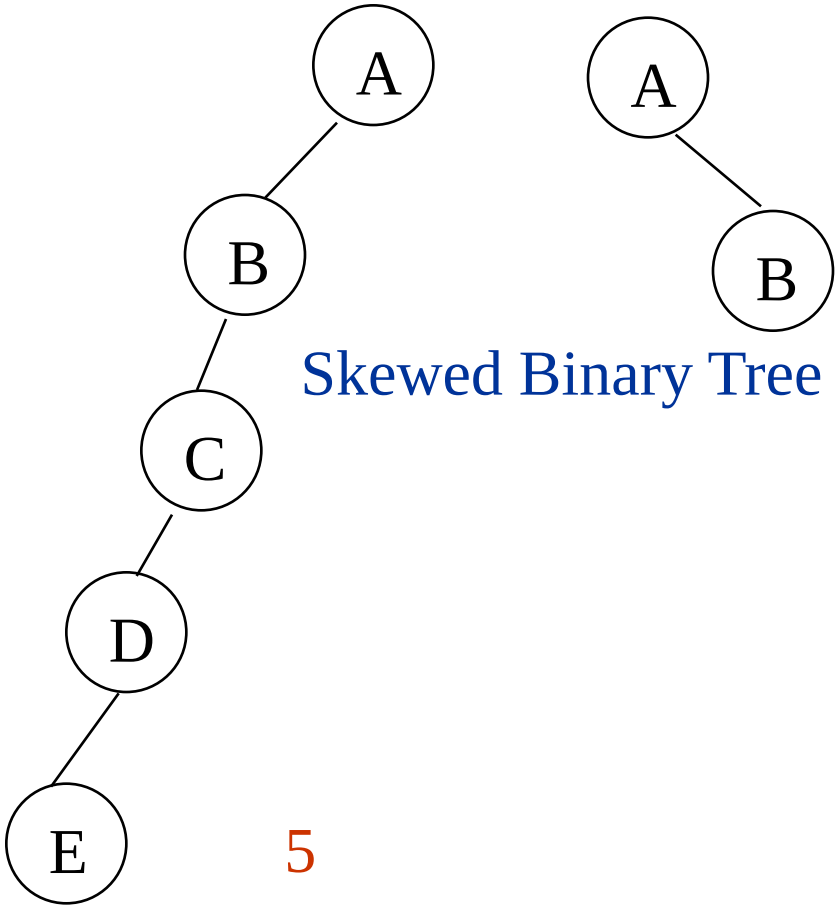
*BinTree* makeBT(*bt1*, *item*, *bt2*)::= return a binary tree
whose left subtree is *bt1*, whose right subtree is *bt2*,
and whose root node contains the data *item*

*Bintree* leftChild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the left subtree of *bt*

*element* data(*bt*)::= if (IsEmpty(*bt*)) return error
else return the data in the root node of *bt*

*Bintree* rightChild(*bt*)::= if (IsEmpty(*bt*)) return error
else return the right subtree of *bt*

# *Samples of Trees*



# *Maximum Number of Nodes in BT*

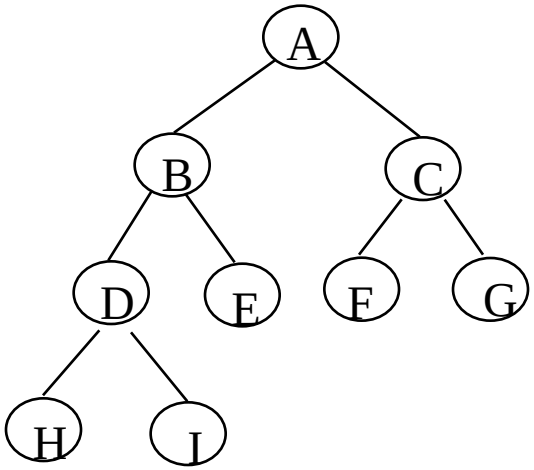
- The maximum number of nodes on level i of a binary tree is $2^{i-1}$, i>=1.
- The maximum nubmer of nodes in a binary tree of depth k is $2^{k}-1$, k>=1.

**Prove by induction.**

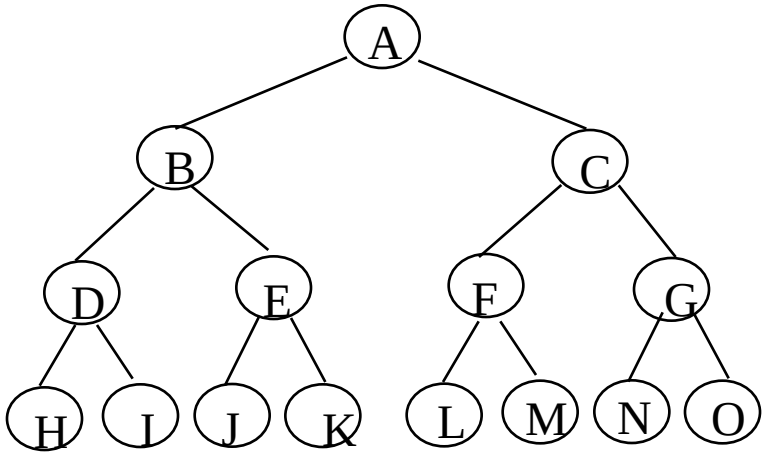
$$\sum_{i=1}^{k} 2^{i-1} = 2^{k} - 1$$

# *Full BT vs. Complete BT*

- A full binary tree of depth *k* is a binary tree of depth *k* having $2^k$-1 nodes, *k*>=0.
- A binary tree with *n* nodes and depth *k* is complete *iff* its nodes correspond to the nodes numbered from 1 to *n* in the full binary tree of depth *k*.




Complete binary tree

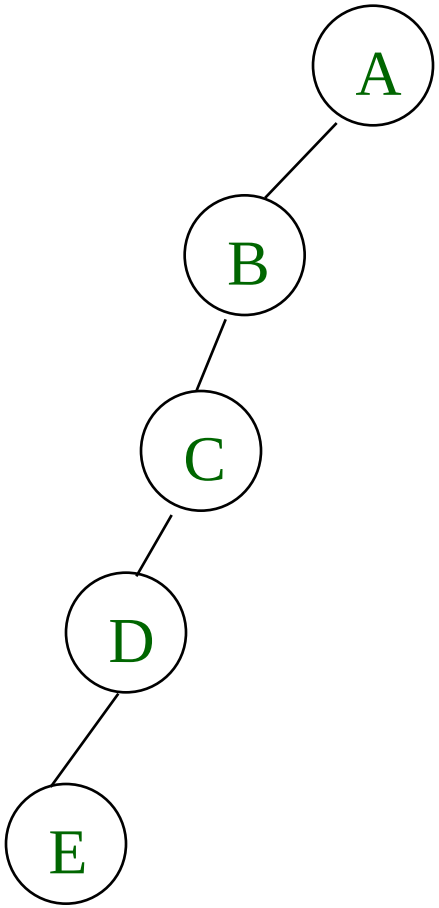



Full binary tree of depth 4

# *Binary Tree Representations*

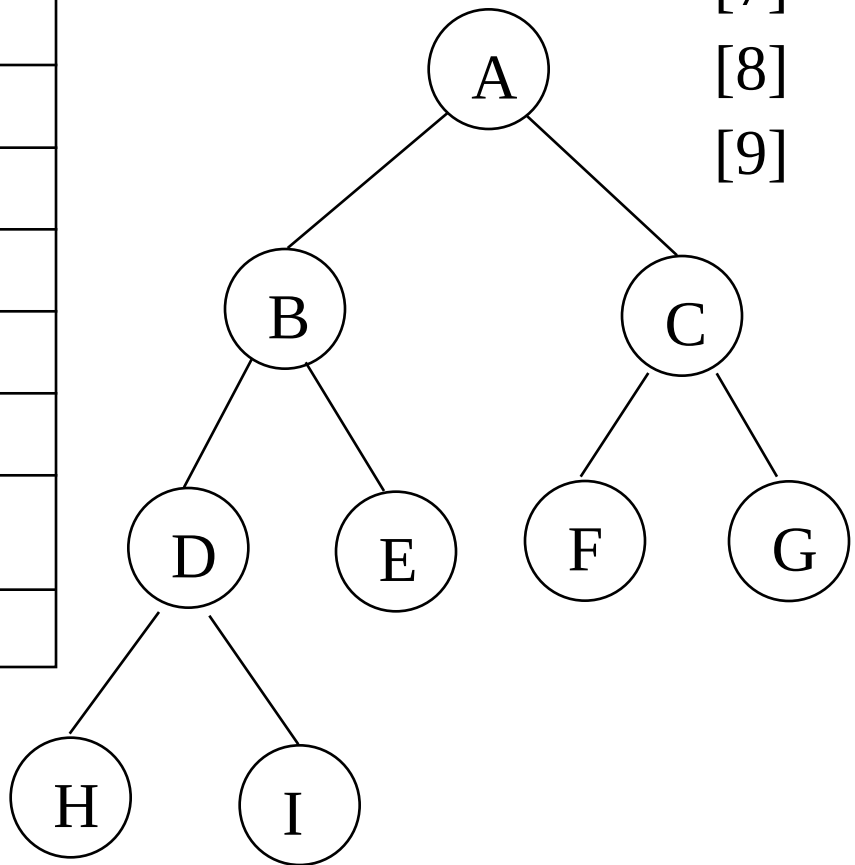
- If a complete binary tree with $n$ nodes (depth = $\log n + 1$) is represented sequentially, then for any node with index $i$, $1 <= i <= n$, we have:
  - *parent*($i$) is at $i/2$ if $i$!=1. If $i$=1, $i$ is at the root and has no parent.
  - *leftChild*($i$) is at $2i$ if $2i <= n$. If $2i$>n, then $i$ has no left child.
  - *rightChild*($i$) is at $2i+1$ if $2i + 1 <= n$. If $2i + 1$ >n, then $i$ has no right child.

# *Sequential Representation*




| | |
|---|---|
| [1] | A |
| [2] | B |
| [3] | -- |
| [4] | C |
| [5] | -- |
| [6] | -- |
| [7] | -- |
| [8] | D |
| [9] | -- |
| . | . |
| [16] | E |

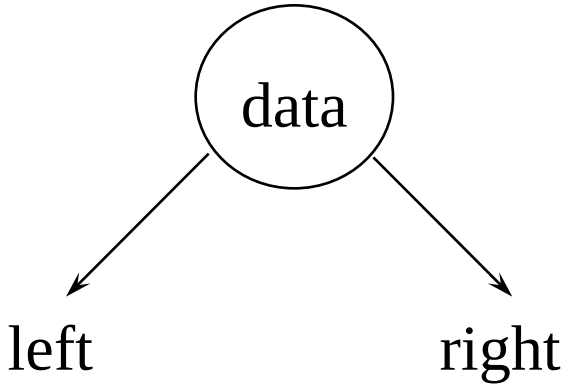
**(1) waste space**
**(2) insertion/deletion problem**




| | |
|---|---|
| [1] | A |
| [2] | B |
| [3] | C |
| [4] | D |
| [5] | E |
| [6] | F |
| [7] | G |
| [8] | H |
| [9] | I |

# *Linked Representation*

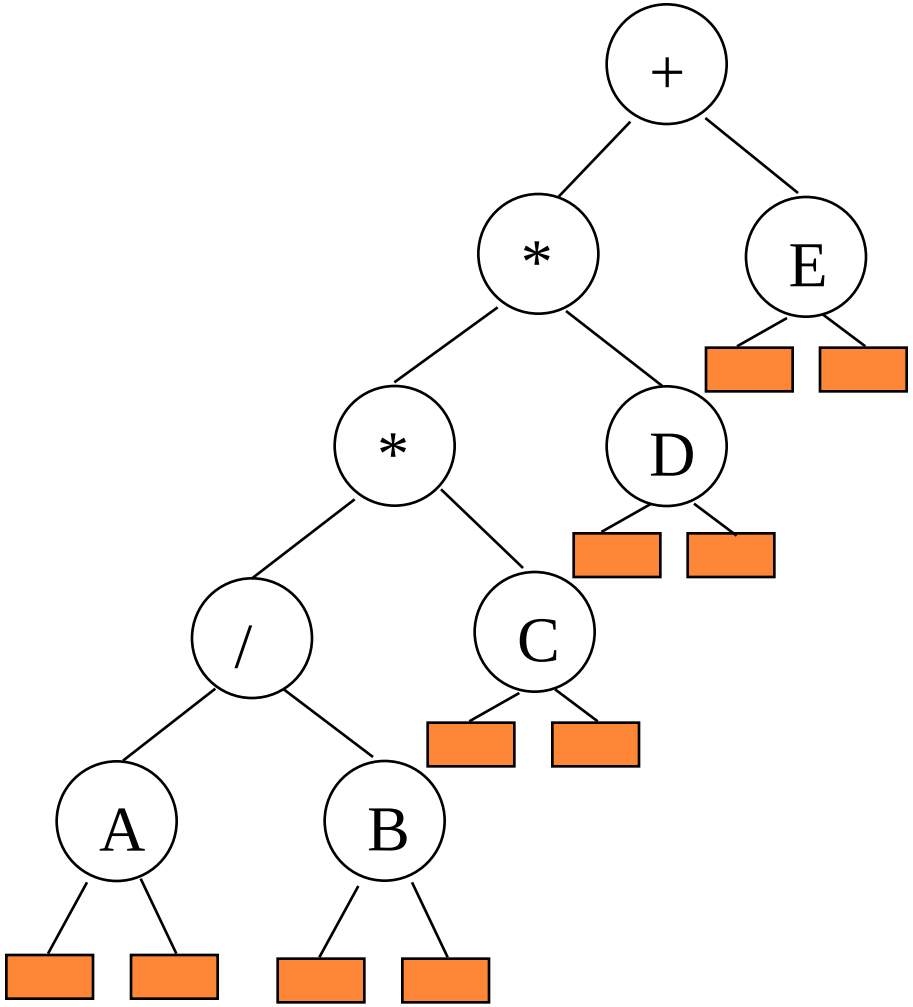
```
typedef struct tnode *ptnode;
typedef struct tnode {
 int data;
 ptnode left, right;
};
```

# *Binary Tree Traversals*

- Let L, V, and R stand for moving left, visiting the node, and moving right.
- There are six possible combinations of traversal
  - lRr, lrR, Rlr, Rrl, rRl, rlR
- Adopt convention that we traverse left before right, only 3 traversals remain
  - lRr, lrR, Rlr
  - inorder, postorder, preorder

# *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

# *Inorder Traversal* *(recursive version)*

```
void inorder(ptnode ptr)
/* inorder tree traversal */
{
     if (ptr) {
          inorder(ptr->left);
          printf("%d", ptr->data);
          indorder(ptr->right);
     }
}
```

A / B * C * D + E

# *Preorder Traversal* *(recursive version)*

```
void preorder(ptnode ptr)
/* preorder tree traversal */
{
     if (ptr) {
          printf("%d", ptr->data);
          preorder(ptr->left);
          predorder(ptr->right);
     }
}
```

+ * * / A B C D E

# *Postorder Traversal* *(recursive version)*

```
void postorder(ptnode ptr)
/* postorder tree traversal */
{
     if (ptr) {
          postorder(ptr->left);
          postdorder(ptr->right);
          printf("%d", ptr->data);
     }
}
```

A B / C * D * E +

# *Level Order Traversal*

*(using queue)*

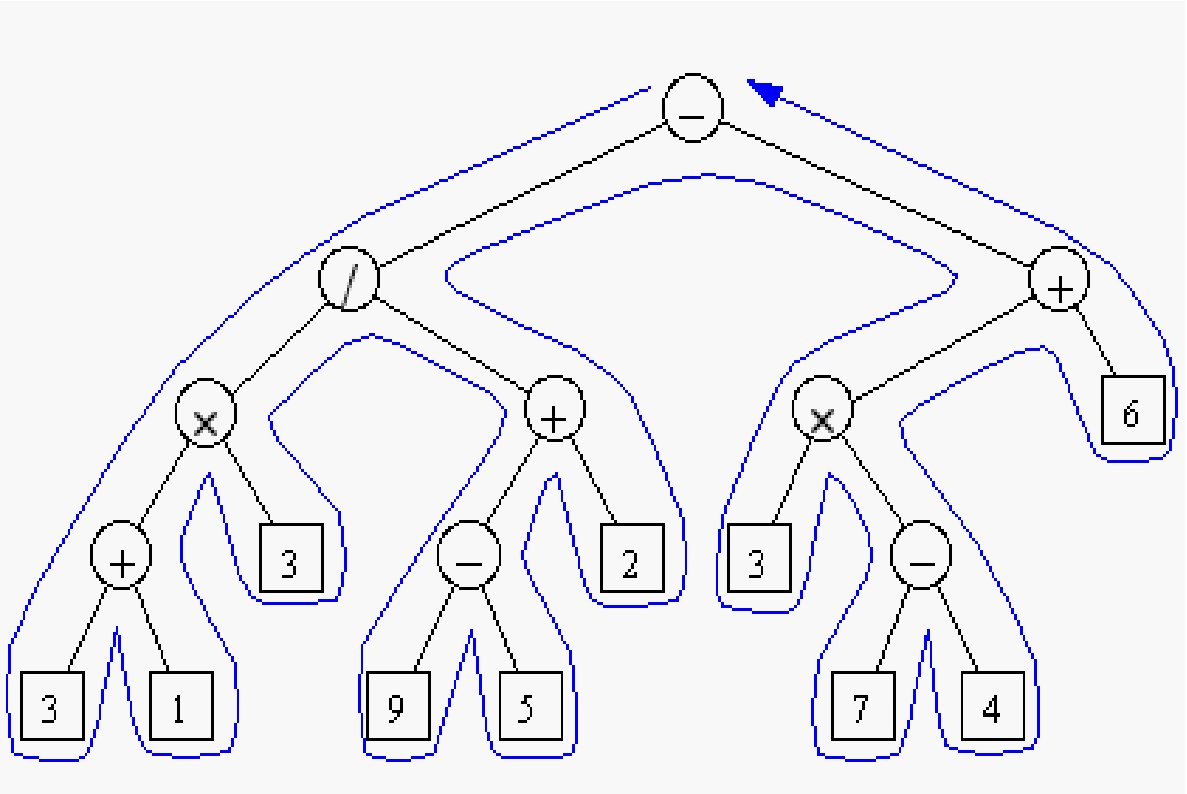
```
void levelOrder(ptnode ptr)
/* level order tree traversal */
{
  int front = rear = 0;
  ptnode queue[MAX_QUEUE_SIZE];
  if (!ptr) return; /* empty queue */
  enqueue(front, &rear, ptr);
  for (;;) {
    ptr = dequeue(&front, rear);
```

```
    if (ptr) {
      printf("%d", ptr->data);
      if (ptr->left)
        enqueue(front, &rear,
                       ptr->left);
      if (ptr->right)
        enqueue(front, &rear,
                       ptr->right);
    }
    else break;
  }
}
```

+ * E * D / C A B

# Euler Tour Traversal

- generic traversal of a binary tree
- the preorder, inorder, and postorder traversals are special cases of the Euler tour traversal
- "walk around" the tree and visit each node three times:
  - on the left
  - from below
  - on the right



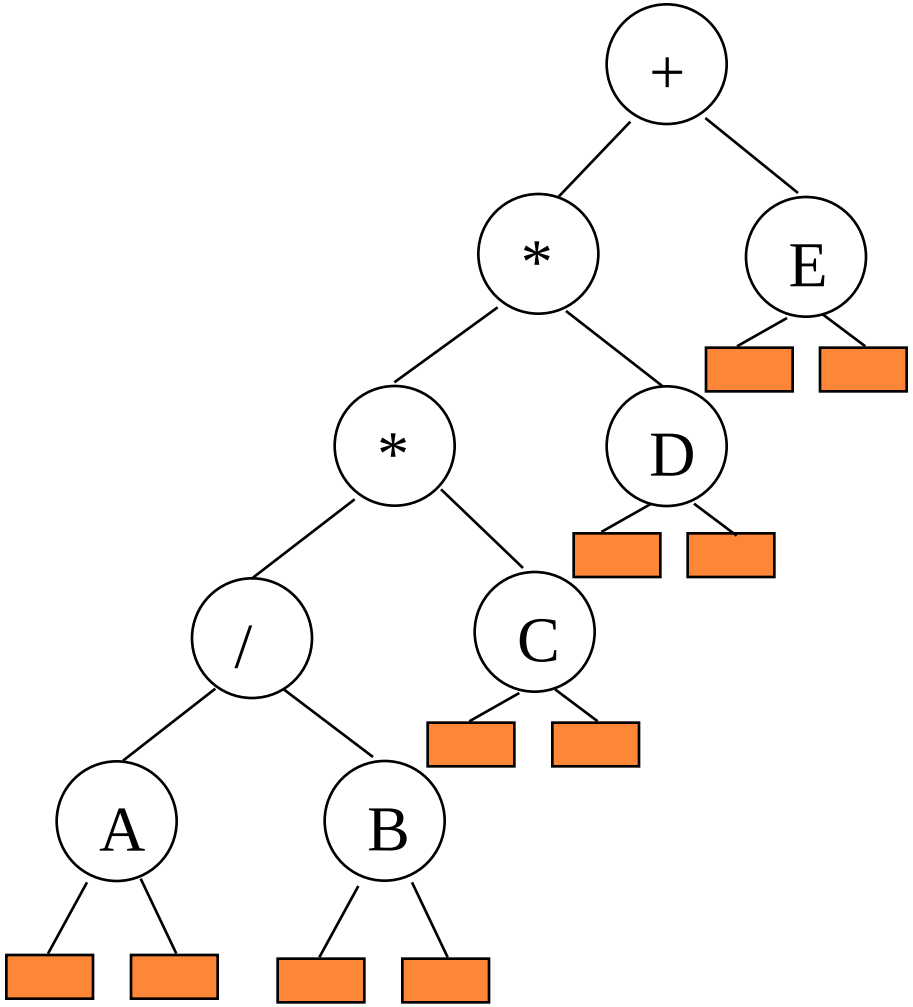
# Euler Tour Traversal (cont'd)

```
eulerTour(node v) {
   perform action for visiting node on the left;
   if v is internal then
        eulerTour(v->left);
   perform action for visiting node from below;
   if v is internal then
        eulerTour(v->right);
   perform action for visiting node on the right;
}
```

# Euler Tour Traversal (cont'd)

- preorder traversal = Euler Tour with a "visit" only on the left
- inorder = ?
- postorder = ?
- Other applications: compute number of descendants for each node v:
  - counter = 0
  - increment counter each time node is visited on the left
  - #descendants = counter when node is visited on the right – counter when node is visited on the left + 1
- Running time for Euler Tour?

# *Application: Evaluation of Expressions*




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

# *Inorder Traversal* *(recursive version)*

```
void inorder(ptnode ptr)
/* inorder tree traversal */
{
     if (ptr) {
          inorder(ptr->left);
          printf("%d", ptr->data);
          inorder(ptr->right);
     }
}
```

A / B * C * D + E

# *Preorder Traversal* *(recursive version)*

```
void preorder(ptnode ptr)
/* preorder tree traversal */
{
     if (ptr) {
          printf("%d", ptr->data);
          preorder(ptr->left);
          preorder(ptr->right);
     }
}
```

+ * * / A B C D E

# *Postorder Traversal* *(recursive version)*

```
void postorder(ptnode ptr)
/* postorder tree traversal */
{
     if (ptr) {
          postorder(ptr->left);
          postorder(ptr->right);
          printf("%d", ptr->data);
     }
}
```

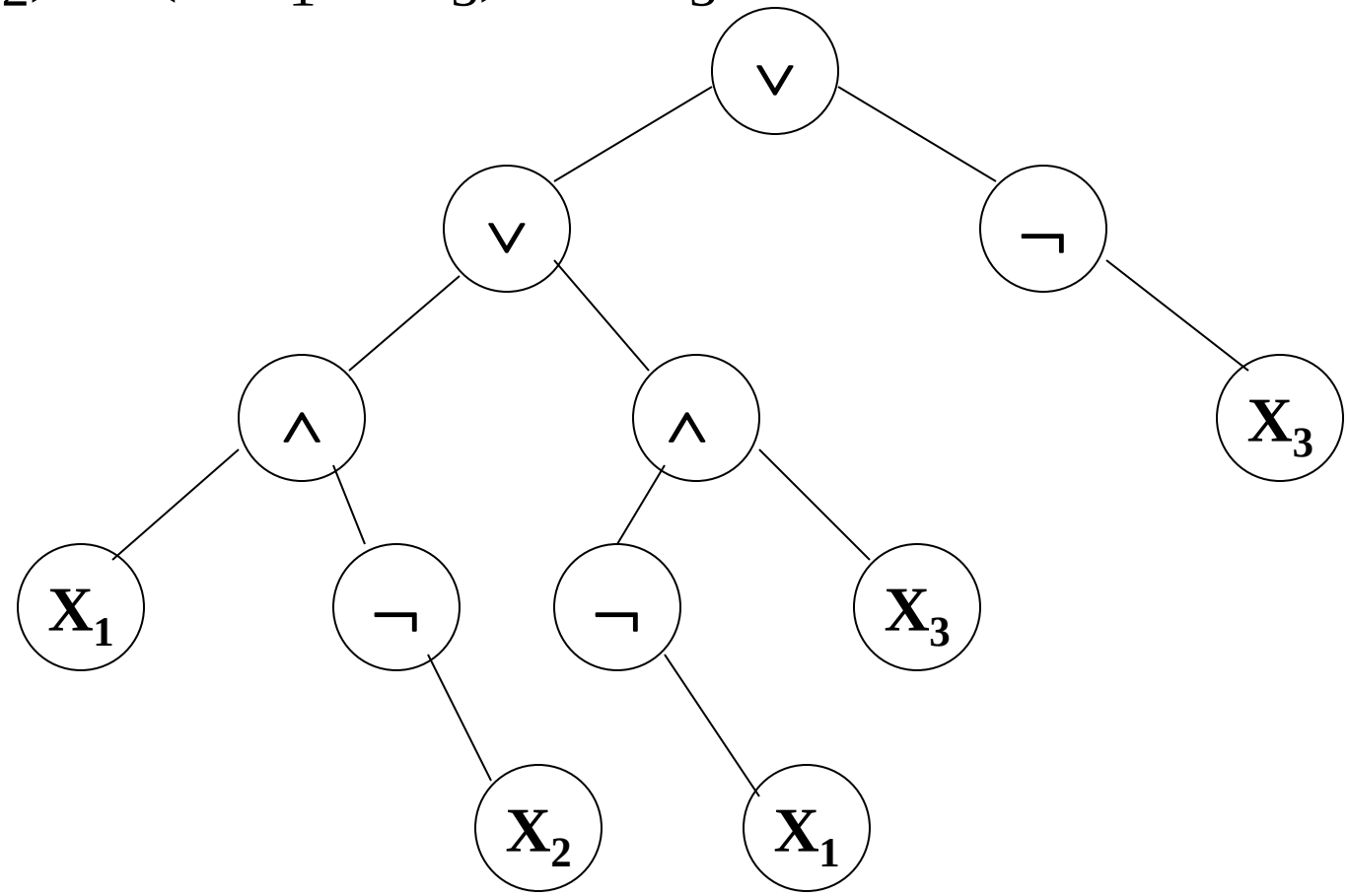
A B / C * D * E +

# Application: Propositional Calculus Expression

- A variable is an expression.
- If x and y are expressions, then ¬x, x∧y, x∨y are expressions.
- Parentheses can be used to alter the normal order of evaluation (¬ > ∧ > ∨).
- Example: $x_1 \vee (x_2 \wedge \neg x_3)$

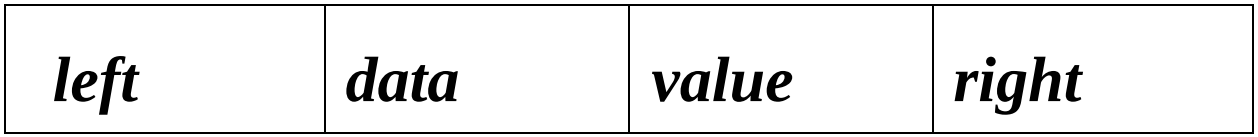
# Propositional Calculus Expression

$$(x_1 \wedge \neg x_2) \vee (\neg\, x_1 \wedge x_3) \vee \neg x_3$$




postorder traversal (postfix evaluation)

# NODE STRUCTURE




```
typedef emun {not, and, or, true, false } logical;
typedef struct tnode *ptnode;
typedef struct node {
        logical         data;
        short int       value;
        ptnode right, left;
        } ;
```

# POSTORDER EVAL

```
void post_order_eval(ptnode node)
{
/* modified post order traversal to evaluate a propositional
calculus tree */
   if (node) {
      post_order_eval(node->left);
      post_order_eval(node->right);
      switch(node->data) {
        case not:  node->value =
              !node->right->value;
              break;
```

# POSTORDER EVAL (CONT'D)

```
        case and:    node->value =
                node->right->value &&
                node->left->value;
                break;
         case or:       node->value =
                node->right->value | |
                node->left->value;
                break;
          case true:    node->value = TRUE;
                break;
          case false:  node->value = FALSE;
        }
    }
}
```