CSE 105:
Data Structures and Algorithms-I
(Part 2)

Instructor

Dr Md Monirul Islam

Graphs and Trees: Representation and Search

Tree Implementation

We have already seen the following binary tree data structure

```
struct BTnode {
    int data;
    struct BTnode *left, *right;
}
```

Tree Implementation

We have already seen the following binary tree data structure

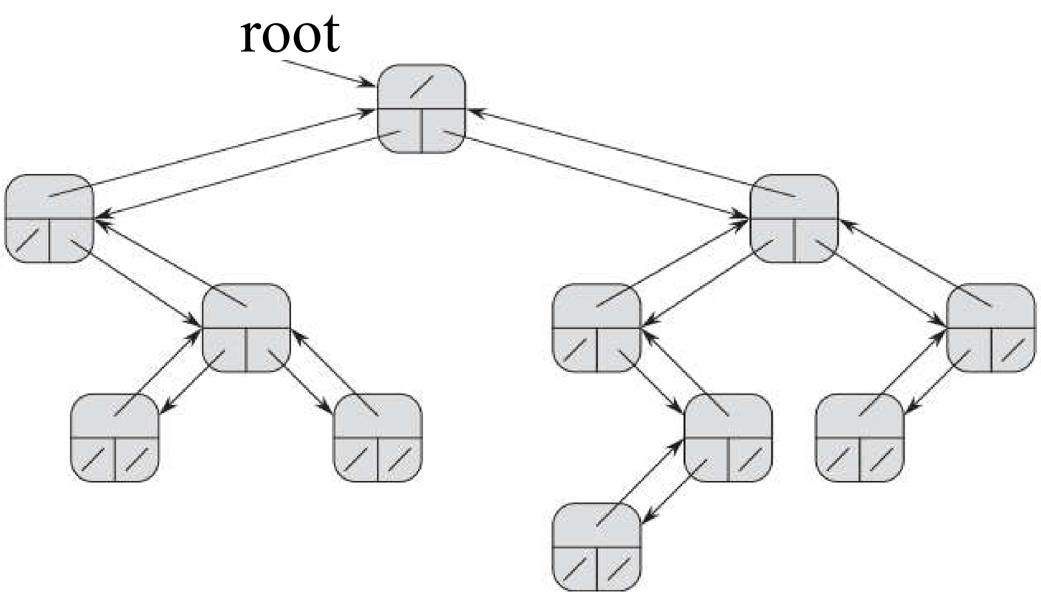
```
struct BTnode {
    int data;
    struct BTnode *left, *right;
}
```

Some applications may require to store **parent** information in the node

```
struct BTnode {
    int data;

    struct BTnode *left, *right, *parent;
}
```

Tree Implementation

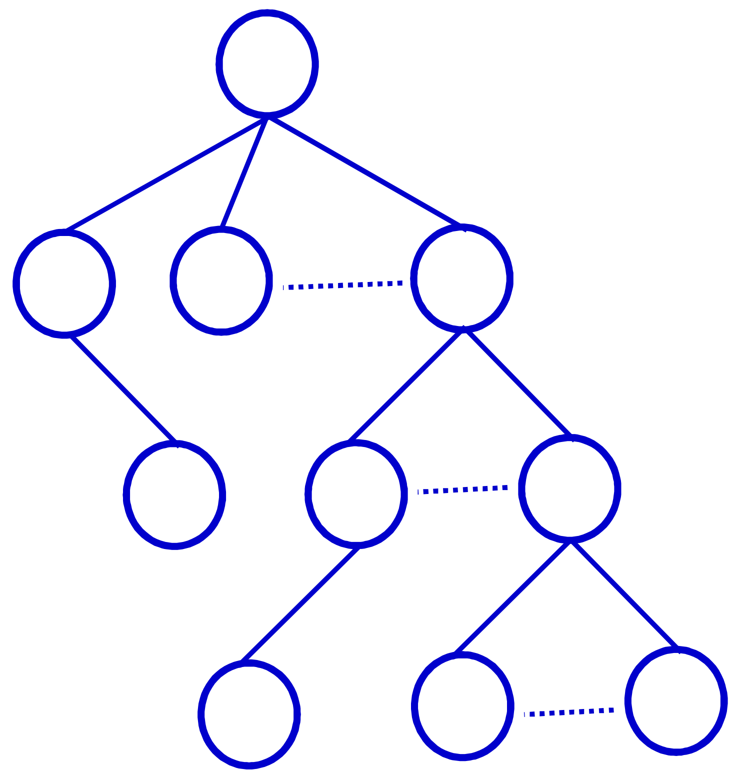


Some applications may require to store **parent** information in the node

```
struct BTnode {
    int data;
    struct BTnode *left, *right, *parent;
}
```

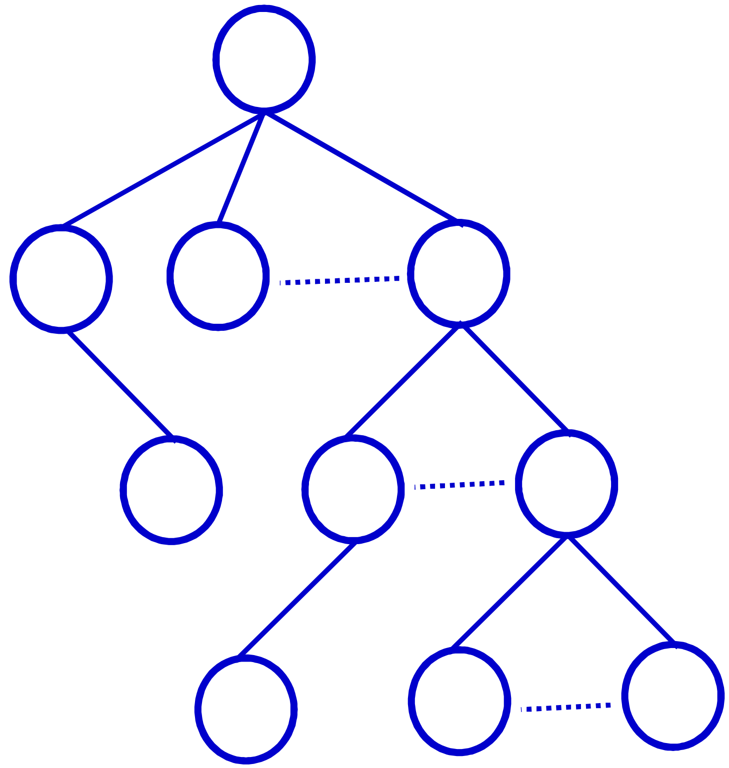
Tree Implementation

If a tree has unbounded number children



Tree Implementation

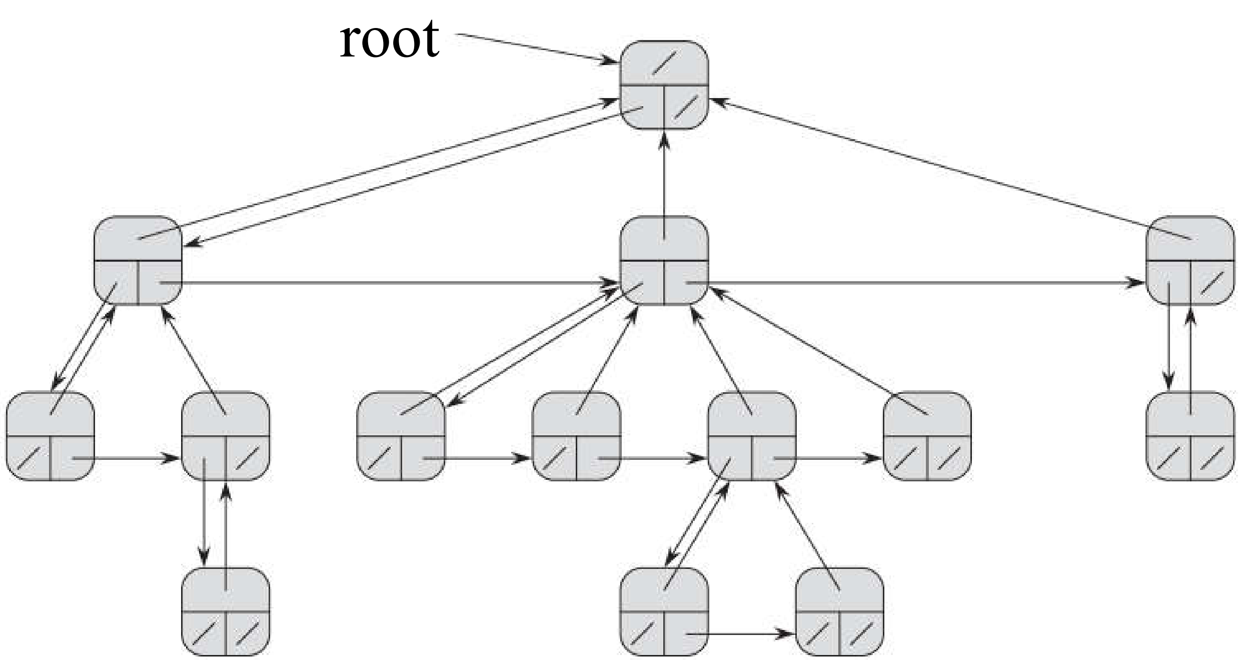
If a tree has unbounded number children



```
struct BTnode {
    int data;
    struct BTnode *left, *rightsibling, *parent;
}
```

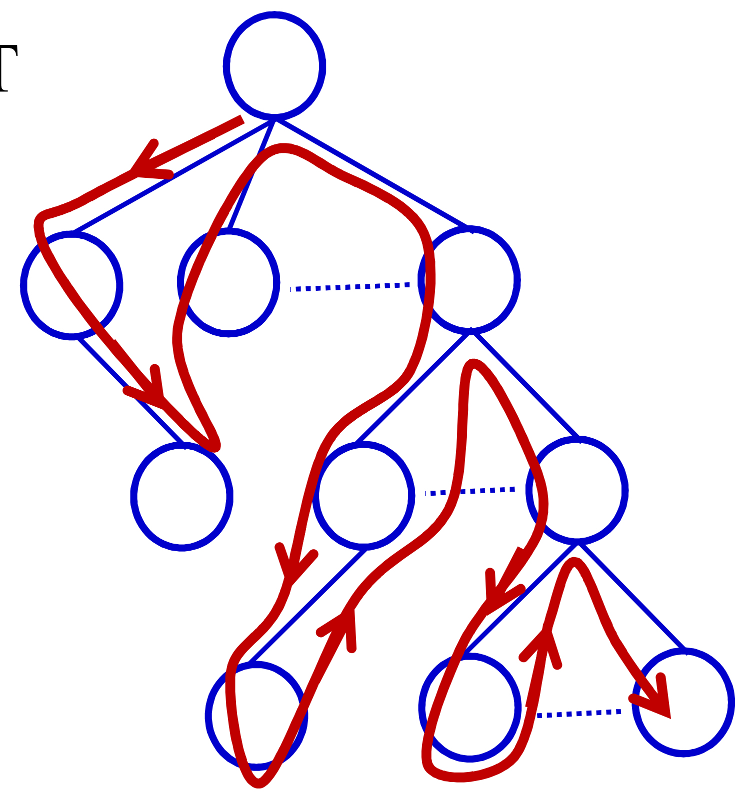
Tree Implementation

```
struct BTnode {
    int data;
    struct BTnode *left, *rightsibling, *parent;
}
```



Tree Traversal

- process for visiting the nodes in some order is called a traversal.
- systematic way of visiting all the nodes of T
- visits the root and travers its subtrees



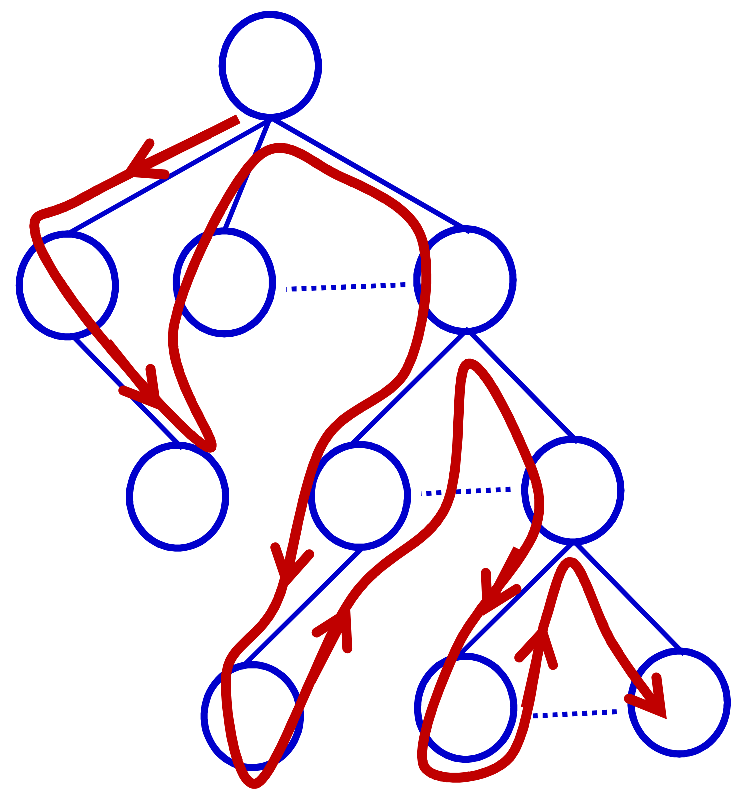
Tree Traversal

3 main traversal methods:

- Preorder Traversal (applicable for any tree)
- Postorder Traversal (applicable for any tree)
- Inorder Traversal (of a binary tree)

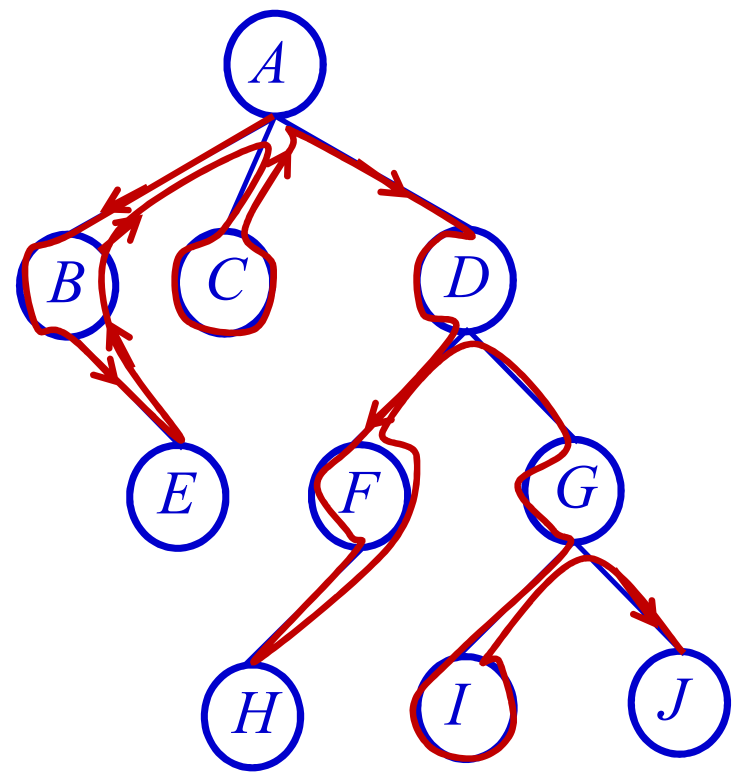
- Other than the above, level order traversal

- Traversing every node exactly once is called an enumeration of the tree's nodes.



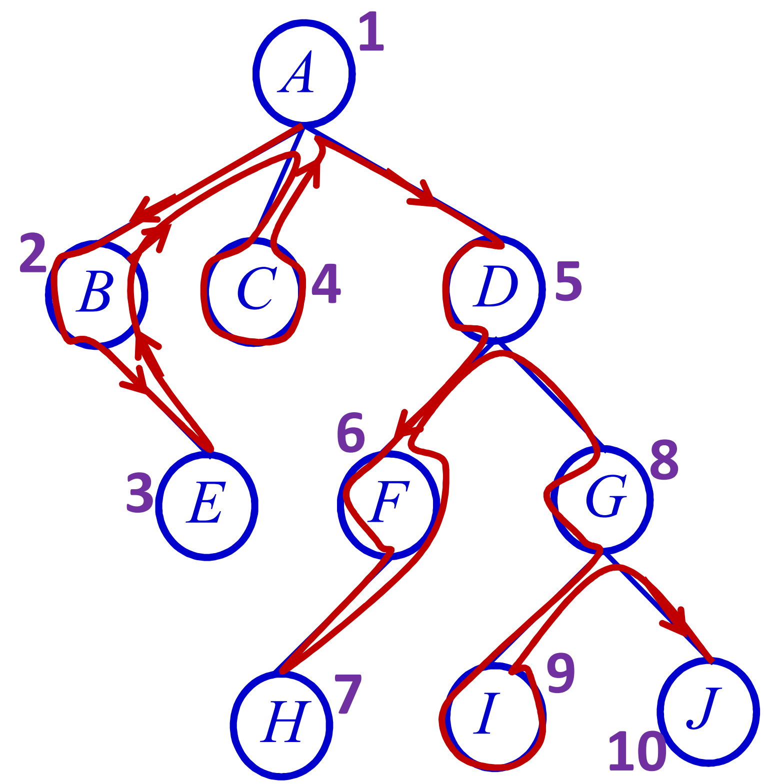
Preorder Tree Traversal

- a node is visited before its descendants
- subtrees are traversed according to the order of the children
- We assume a left to right order



Preorder Tree Traversal

- a node is visited before its descendants
- subtrees are traversed according to the order of the children
- We assume a left to right order

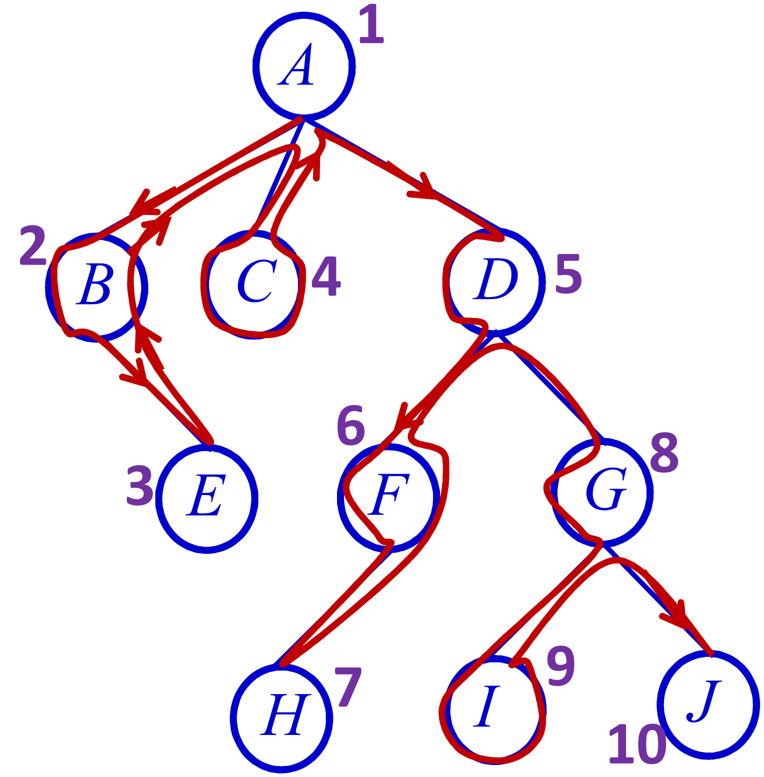


Traversal: A B E C D F H G I J

Preorder Tree Traversal

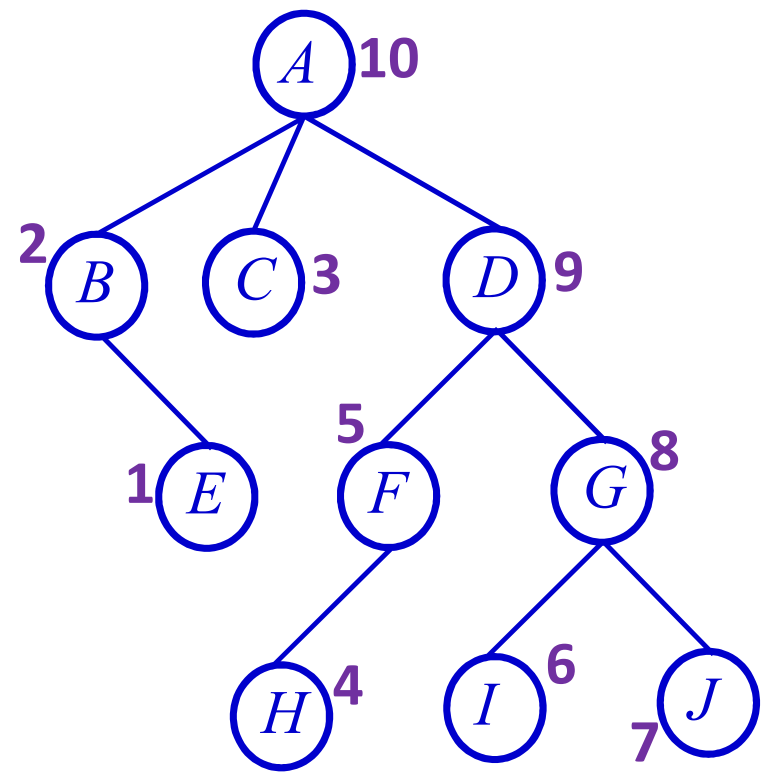
- a node is visited before its descendants
- subtrees are traversed according to the order of the children
- We assume a left to right order

```
Algorithm preOrder(v)
    If v is NULL, return
    visit(v)
    for each child w of v
        preOrder (w)
```



Postorder Tree Traversal

- a node is visited only after all its descendants are visited

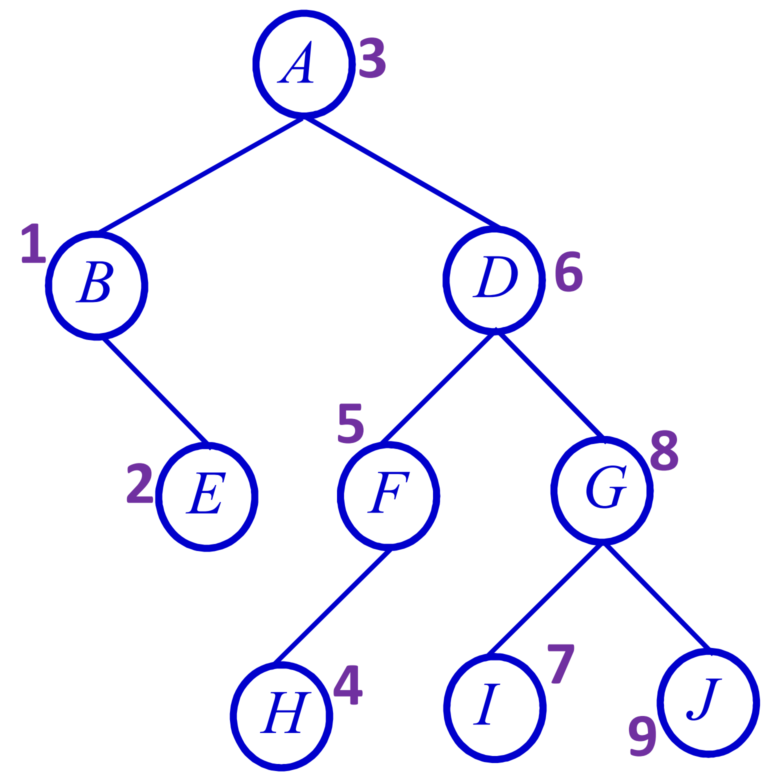


Algorithm ***postOrder*(*v*)**
  If *v* is NULL, return
  for each child ***w*** of ***v***
    postOrder (***w***)
  ***visit*(*v*)**

Traversal: E B C H F I J G D A

Inorder Tree Traversal

- Only for binary tree
- a node is visited *after* its left branch and *before* all its right branch are visited



Algorithm ***inOrder***(*v*)

  If *v* is NULL, return

  inOrder(leftChild(*v*))

  visit(*v*)

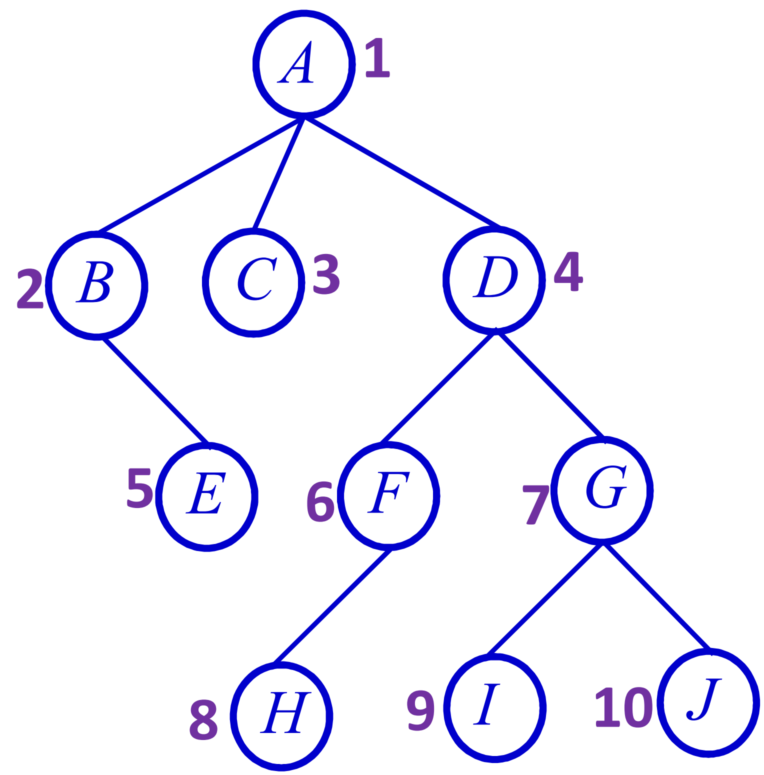
  inOrder(rightChild(*v*))

Traversal: B E A H F D I G J

Level order Tree Traversal

- Nodes are visited level by level from left to right
- Nodes at level i are visited before nodes at level $i + 1$

Traversal: A B C D E F G H I J



Preorder Tree Traversal Code for Binary Tree

We have already seen the following binary tree data structure

```
struct BTnode {
    int data;
    struct BTnode *left, *right;
}
```

Preorder Tree Traversal Code for Binary Tree

We have already seen the following binary tree data structure

```
struct BTnode {
    int data;
    struct BTnode *left, *right;
}

struct BTnode *root;
```

```
/* Recursive Algorithm */
void preorder(struct BTnode *rt)
{
  if (rt == NULL) return; // Empty subtree
  visit_and_doSomething(rt);
  preorder(rt->left);
  preorder(rt->right);
}
```

Inorder Tree Traversal Code for Binary Tree

```
struct BTnode {
    int data;
    struct BTnode *left, *right;
}

struct BTnode *root;
```

```
/* Recursive Algorithm */
void inorder(struct BTnode *rt)
{
  if (rt == NULL) return; // Empty subtree
  inorder(rt->left);
  visit_and_doSomething(rt);
  inorder(rt->right);
}
```

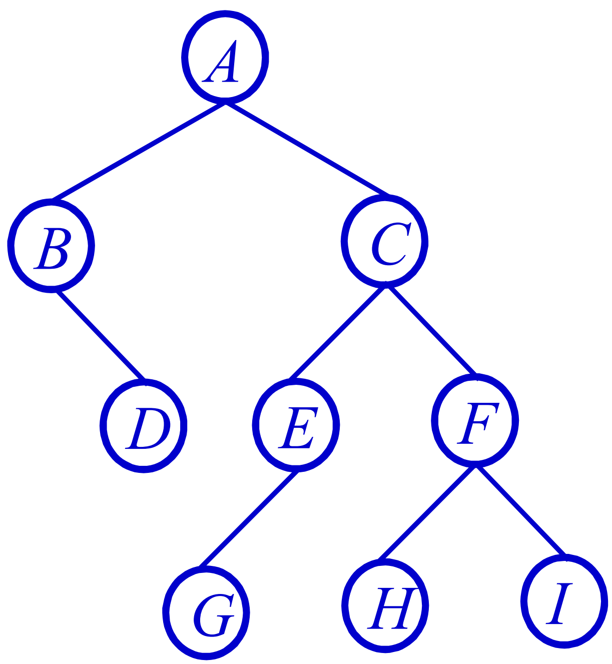
Postorder Tree Traversal Code for Binary Tree

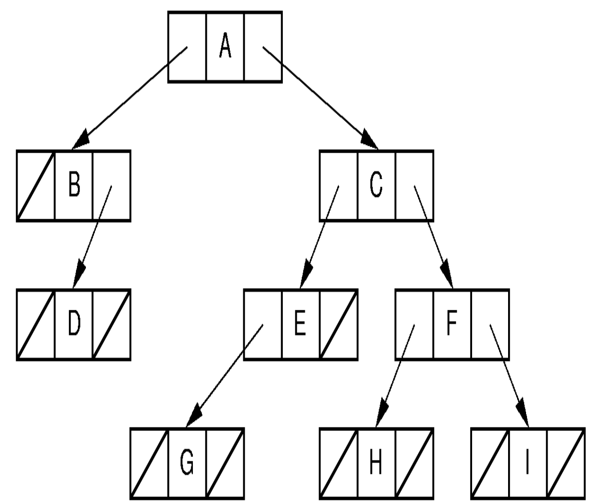
```
/* Recursive Algorithm */
void postorder(struct BTnode *rt)
{
  if (rt == NULL) return; // Empty subtree
  postorder(rt->left);
  postorder(rt->right);
  visit_and_doSomething(rt);

}
```

Binary Tree Implementation Issues

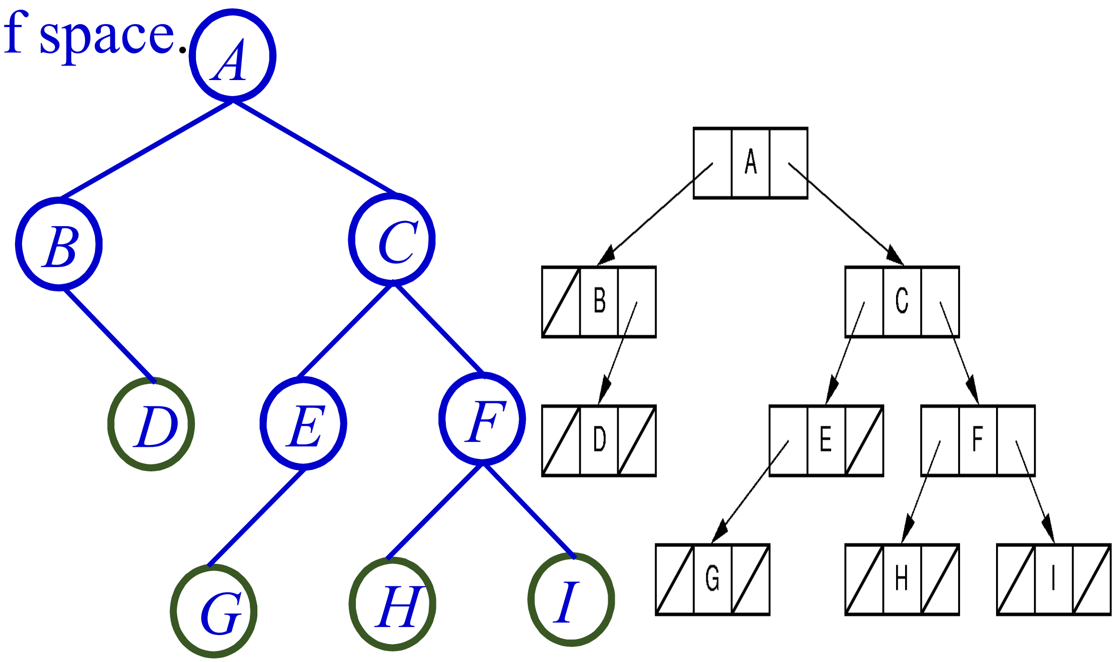
```
struct BTnode {
    int data;
    struct BTnode *left, *right;
}
```





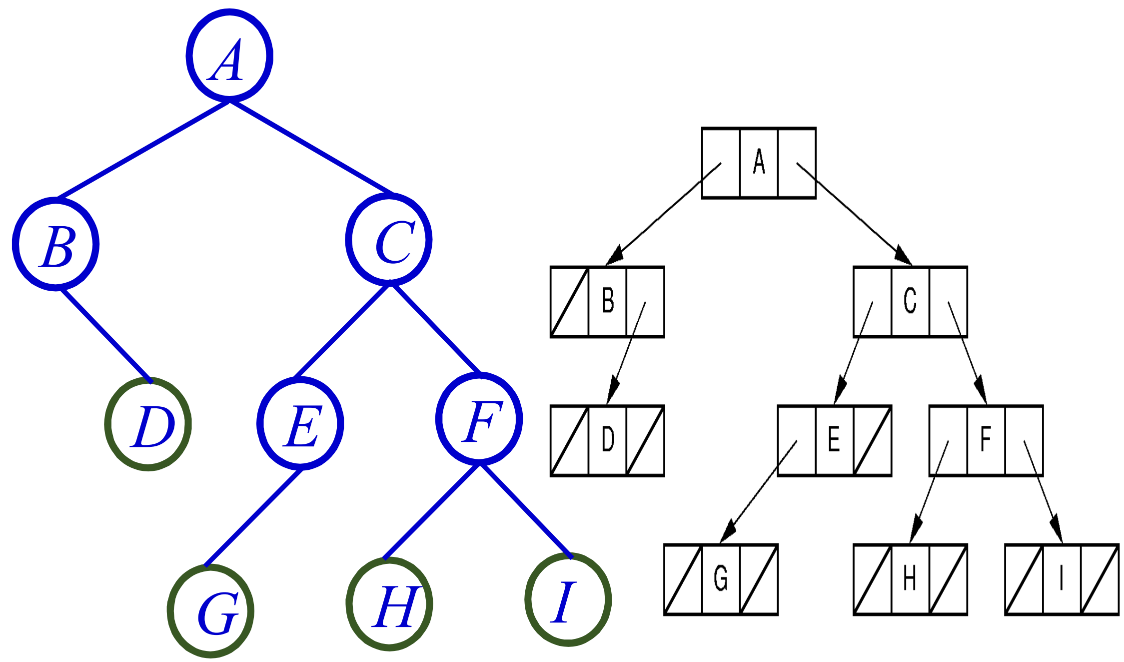
Binary Tree Implementation Issues

- Same class/structure for all leaves and internal nodes.
  - Using the same class for both will simplify the implementation,
  - but might be an inefficient use of space.



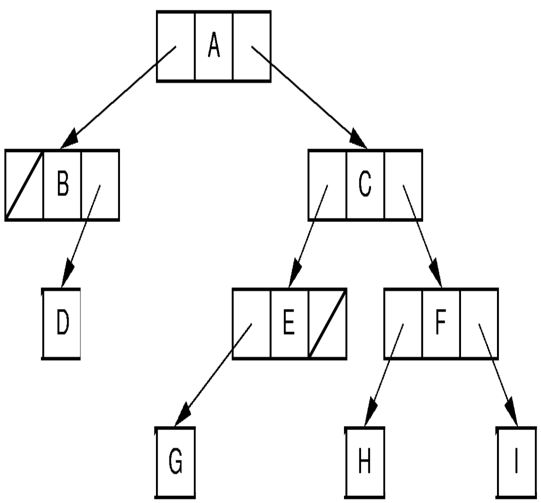
Binary Tree Implementation Issues

- Some applications require data values only for the leaves.
- Other applications require one type of value for the leaves and another for the internal nodes.

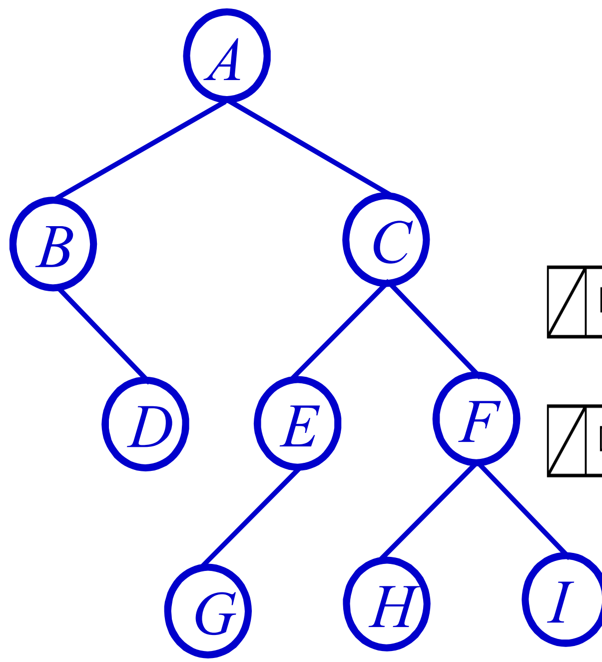


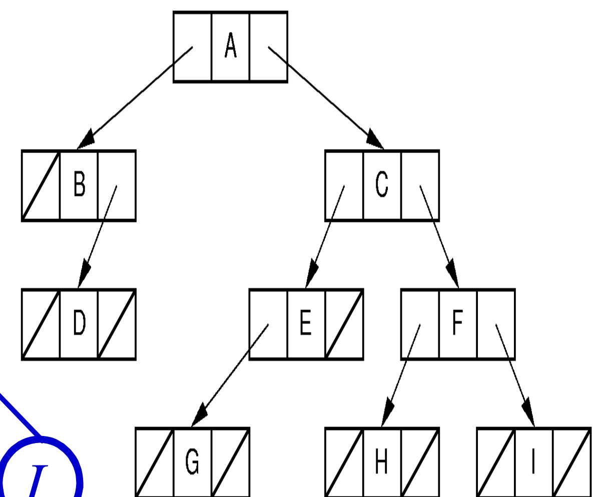
Binary Tree Implementation Issues

- Some applications require data Also, it seems wasteful to store child pointers in the leaf nodes.

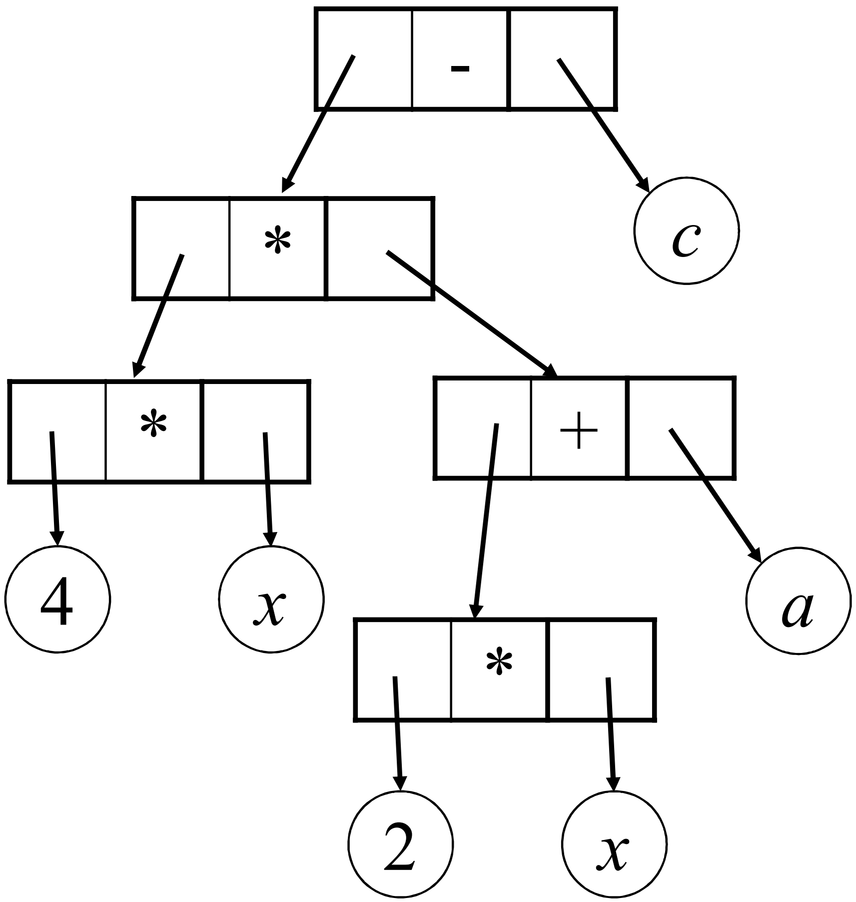


NO child pointer in leaves





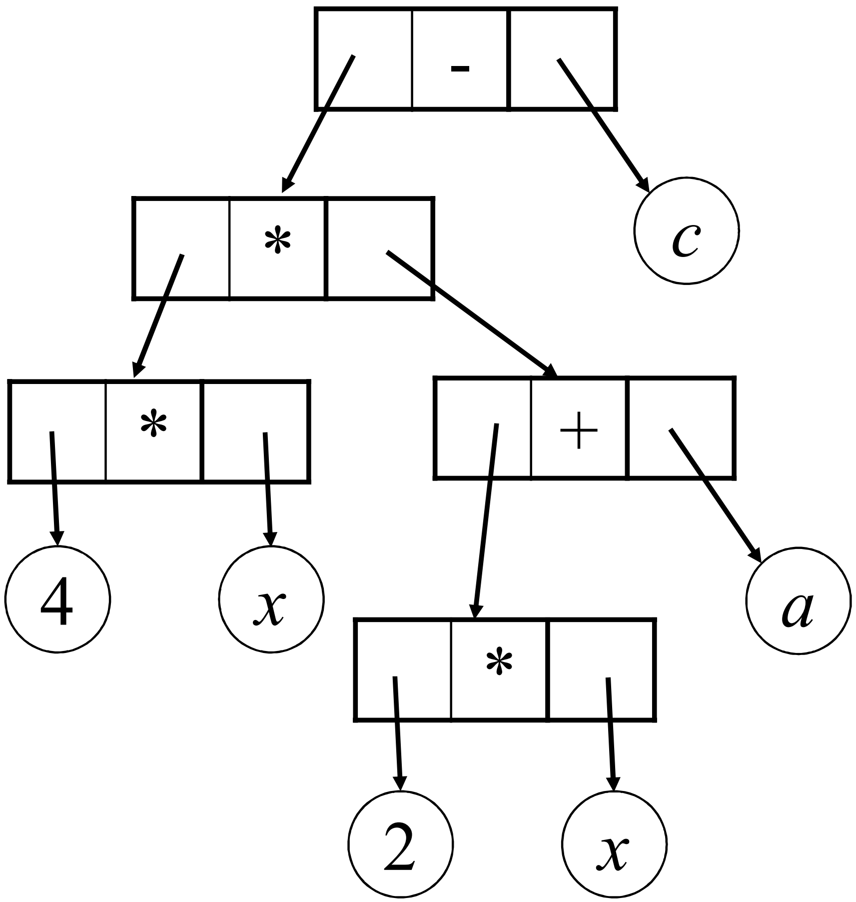
Binary Tree Implementation Issues



$4x\,(2x + a)$ - c

$4 * x \;\; * (2 * x + a)$ - c

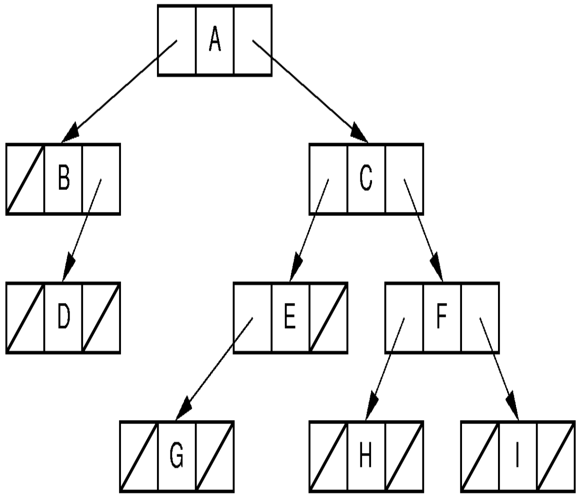
Binary Tree Implementation Issues



- Internal nodes store operators
  - could store a small code identifying the operator (a single byte for the operator's symbol)
- the leaves store operands
  - i.e., variable names or numbers, (considerably larger in order to handle the wider range of possible values)
  - No child pointers though

Space Analysis for Binary Tree Implementation

- Every node has two pointers to its children
- total space = $n(2P + D)$ for a tree of n nodes
  - P: space required by a pointer
  - D: amount of space required by a data value
- So, total overhead: $2Pn$
- Overhead fraction: $2P/(2P+D)$
- $P = D$ => $2/3^{rd}$ of its total space is overhead



Space Analysis for Binary Tree Implementation

- $P = D$ => 2/3rd of its total space is overhead
- From the Full Binary Tree Theorem: Half of the pointers are **null**.
  - half of the pointers are "wasted" **NULL values that serve only to indicate tree** structure, but which do not provide access to new data.