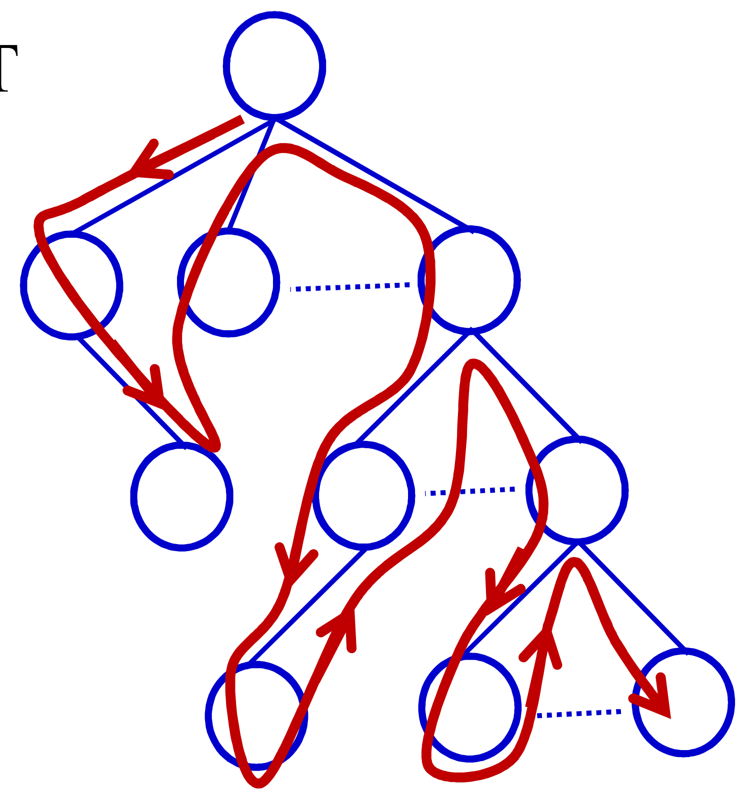
CSE 105:
Data Structures and Algorithms-I
(Part 2)

Instructor

Dr Md Monirul Islam

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



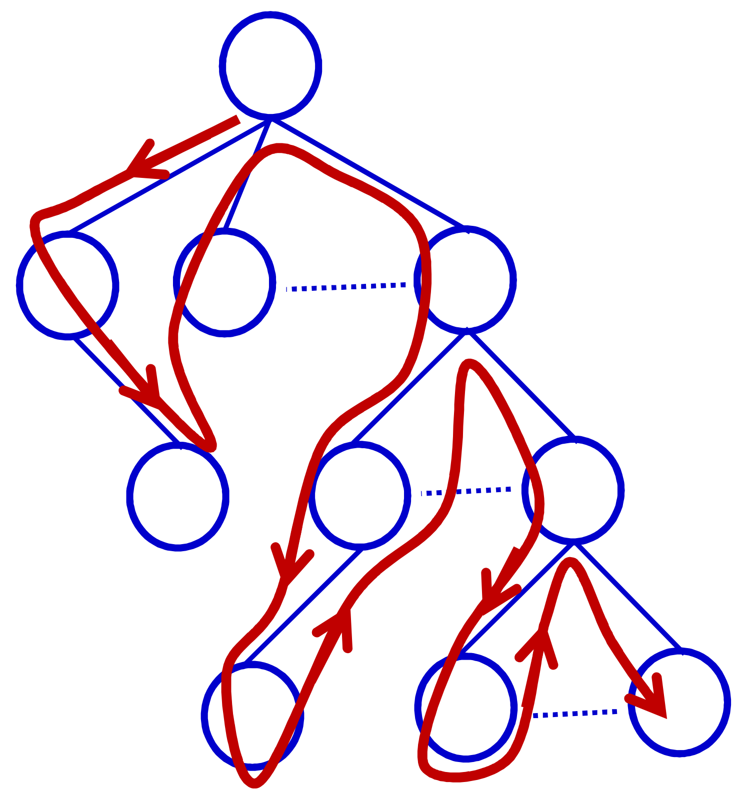
Tree Traversal

3 main traversal methods:

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

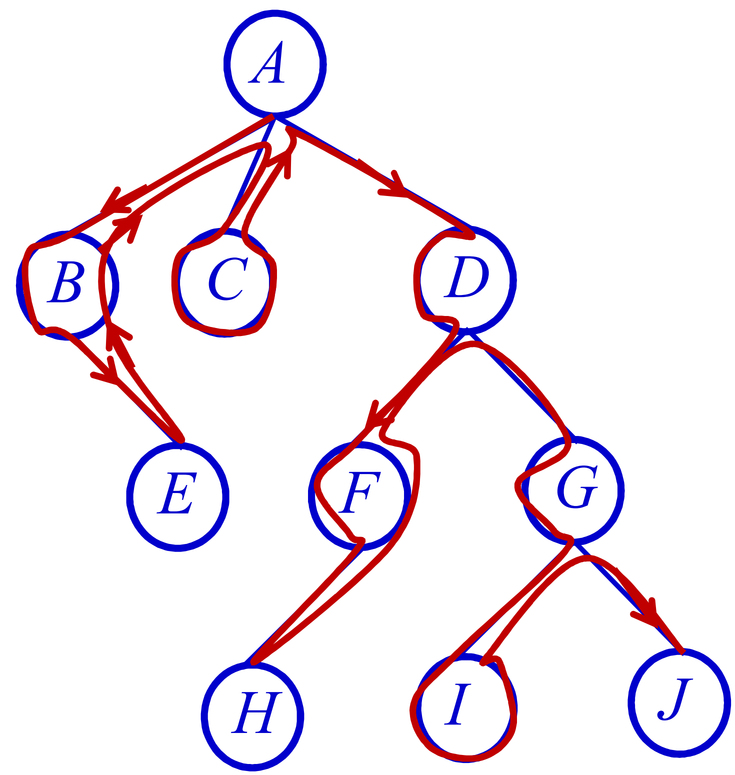
- Other than the above, level order traversal

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



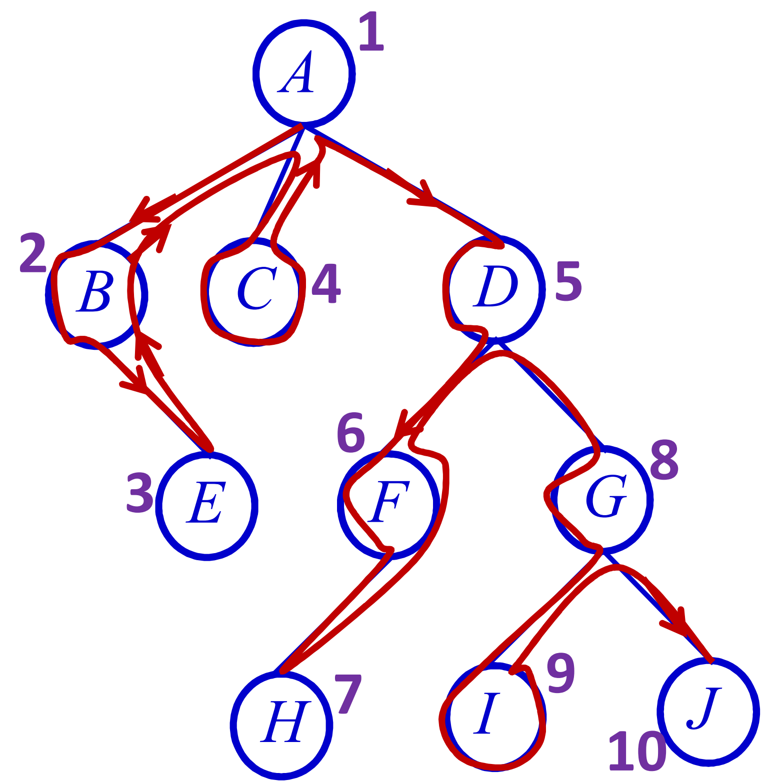
Preorder Tree Traversal

- 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

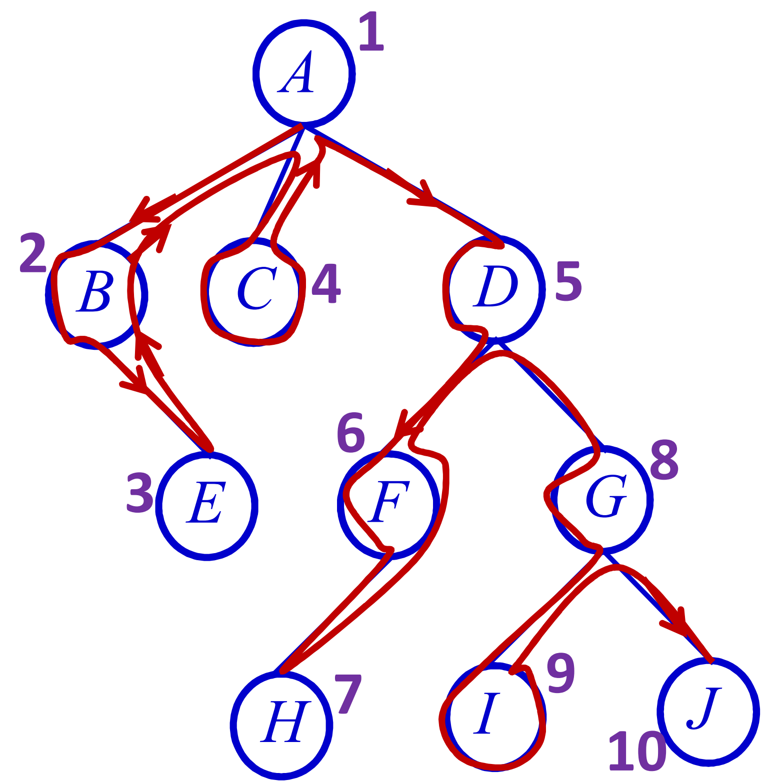


Traversal: A B E C D F H G I J

Preorder Tree Traversal

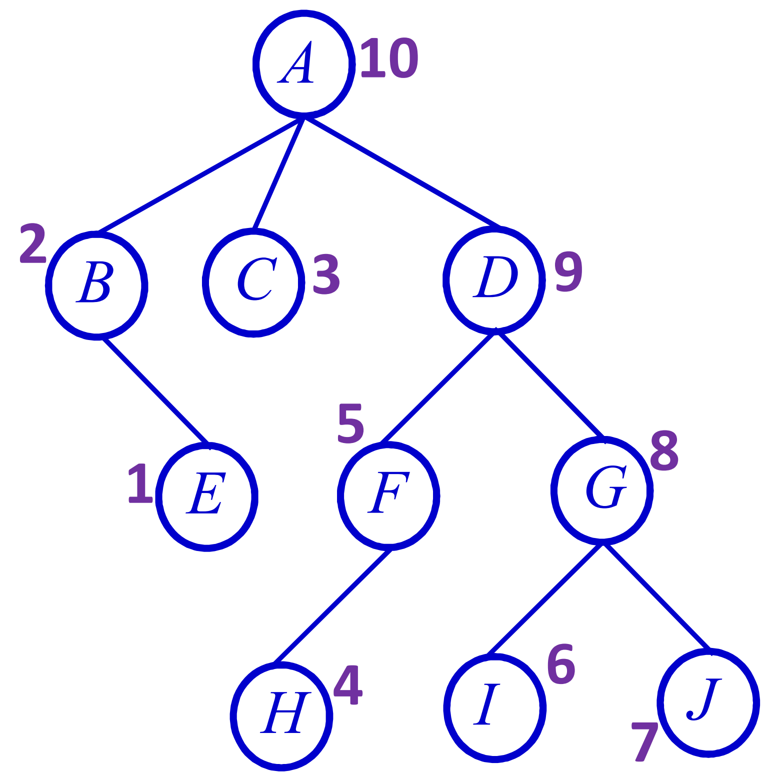
- a node is visited before its descendants
- subtrees are traversed according to the order of the children
- We assume a left to right order

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***

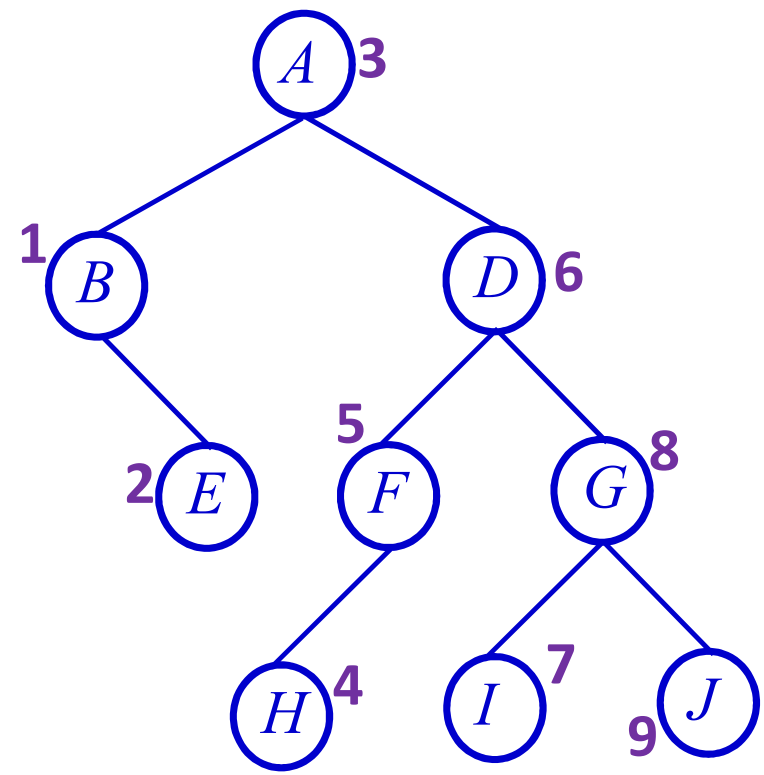
postOrder **(*w*)**

***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***))

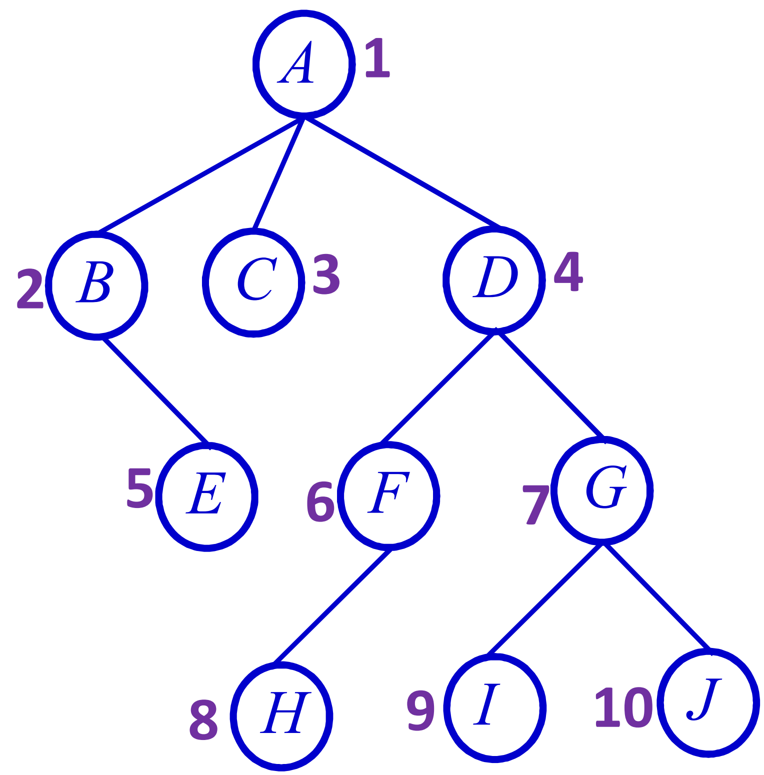
visit(***v***)

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);
}
```

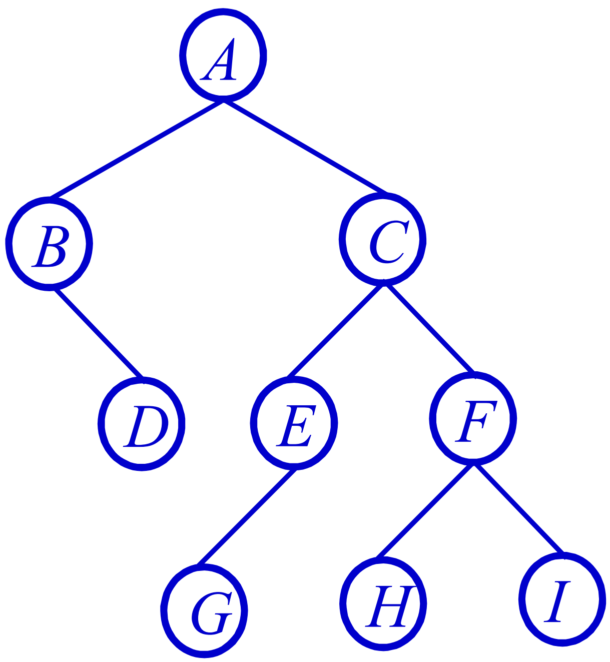
Postorder Tree Traversal Code for Binary Tree

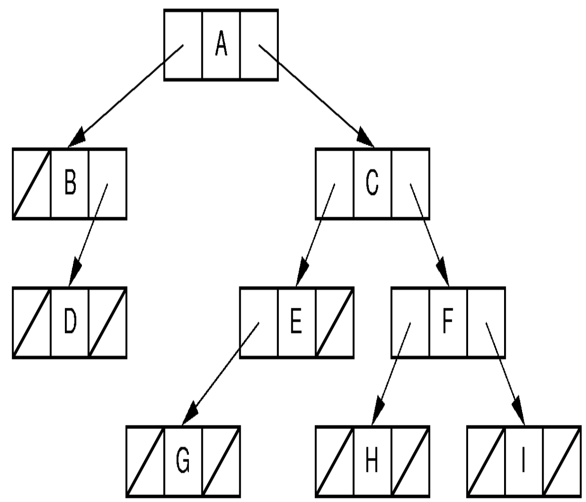
```
/* Recursive Algorithm */
void postorder(struct BTnode *rt)
{
 if (rt == NULL) return; // Empty subtree
 postorder(rt->left);
 postorder(rt->right);
 visit_and_doSomething(rt);

}
```

Binary Tree Implementation Issues

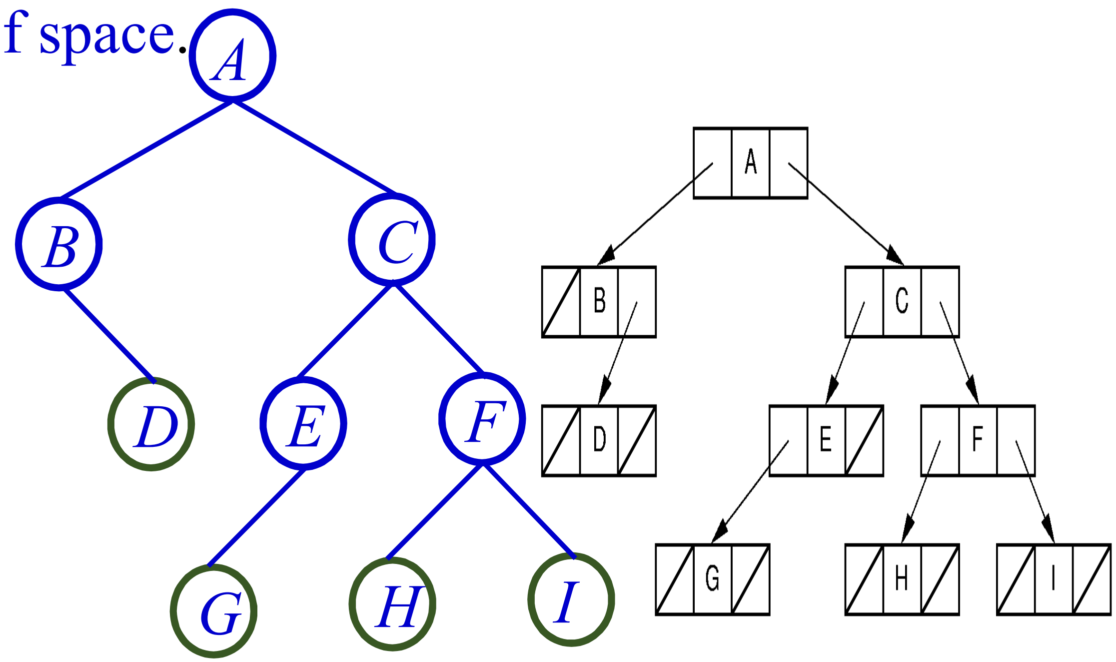
```
struct BTnode {
    int data;
    struct BTnode *left, *right;
}
```





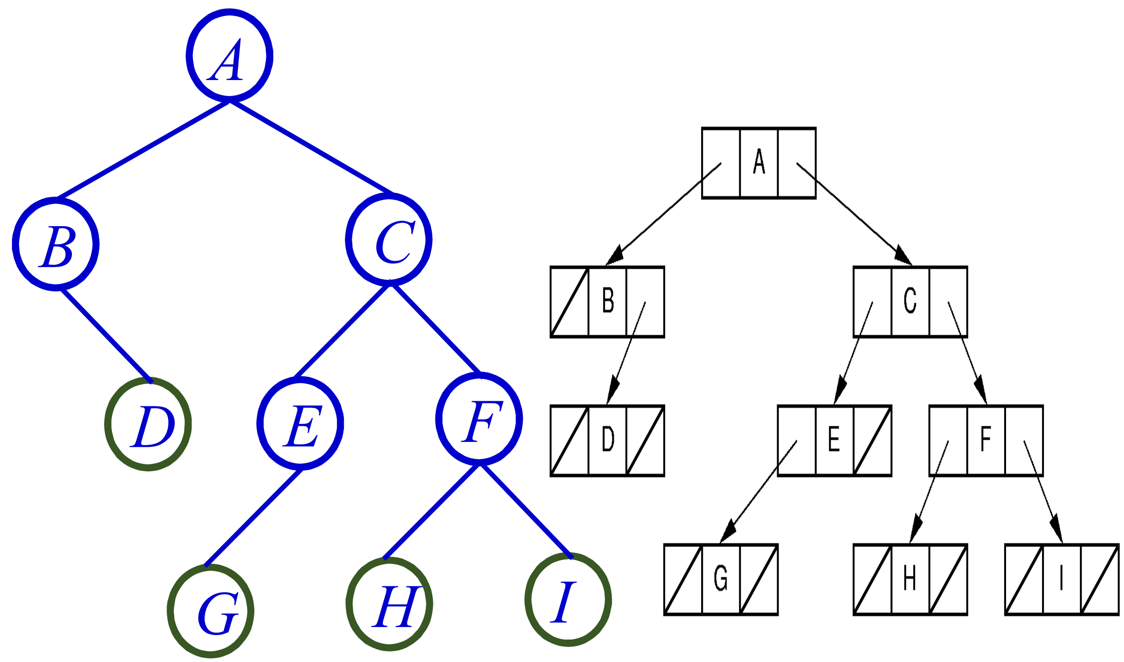
Binary Tree Implementation Issues

- 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.



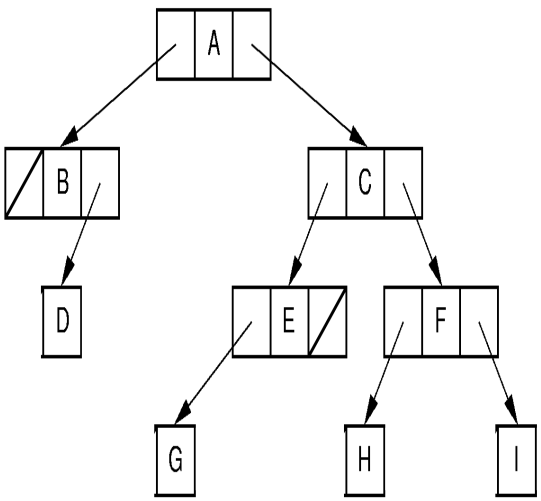
Binary Tree Implementation Issues

- 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.

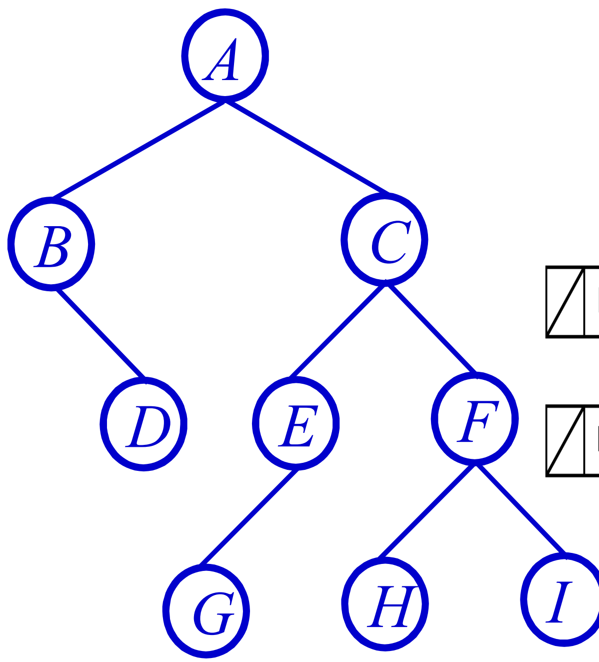


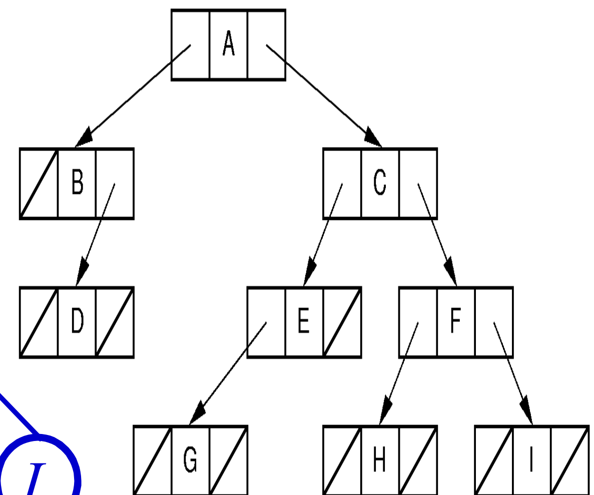
Binary Tree Implementation Issues

- Also, it seems wasteful to store child pointers in the leaf nodes.

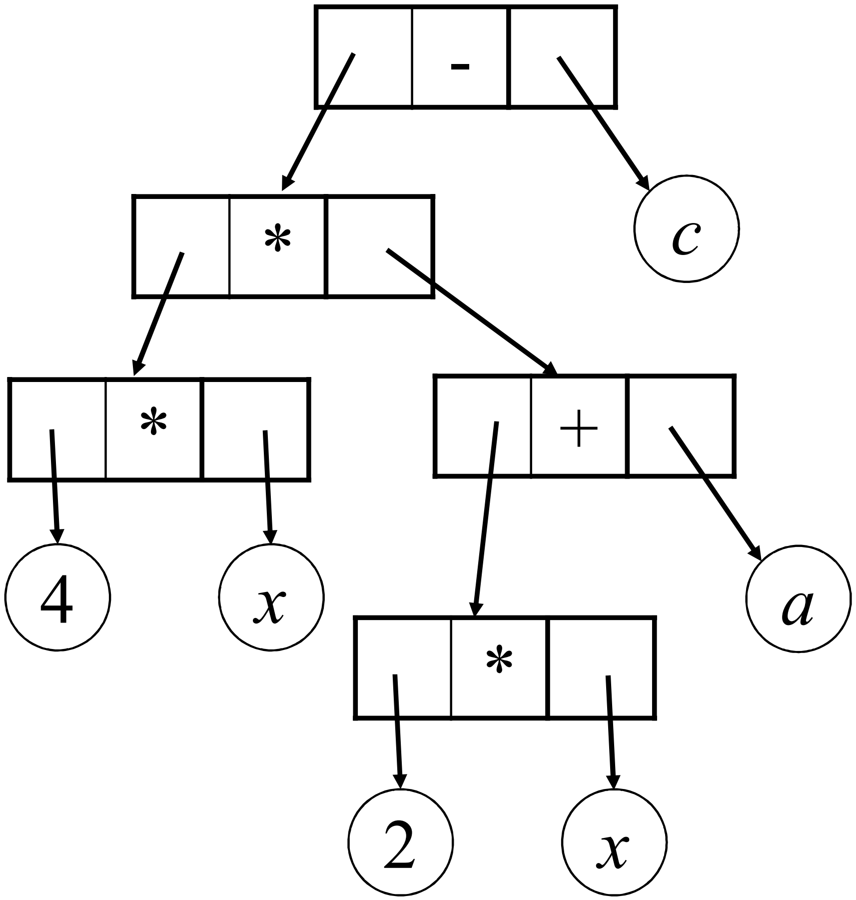


NO child pointer in leaves





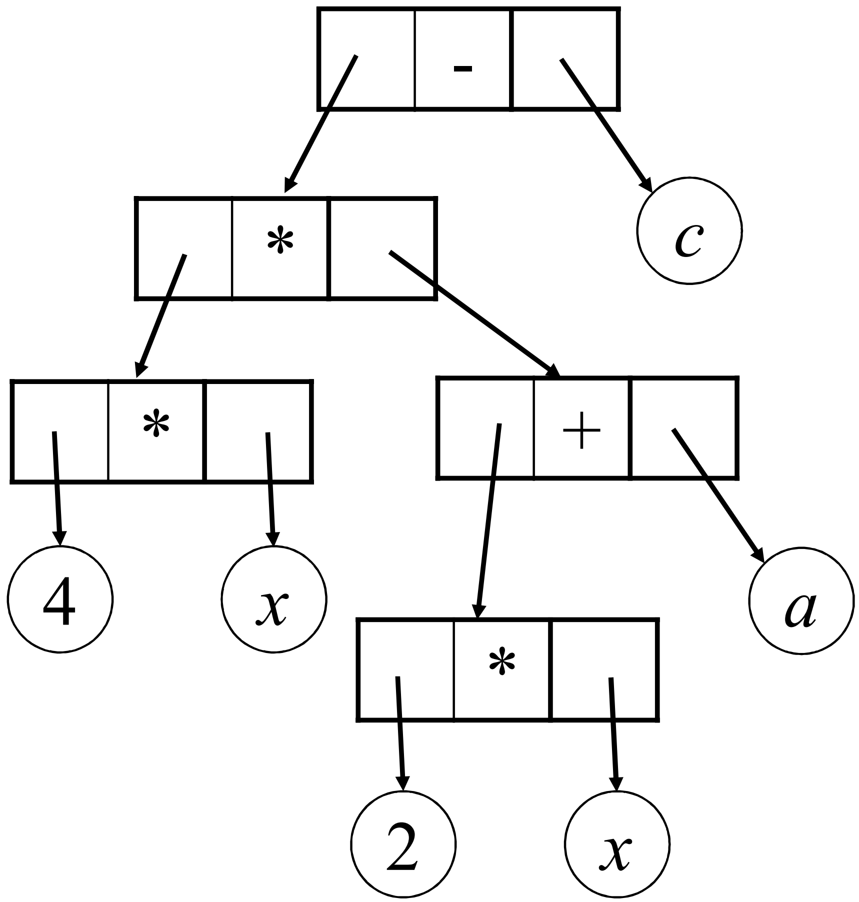
Binary Tree Implementation Issues



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

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

Binary Tree Implementation Issues

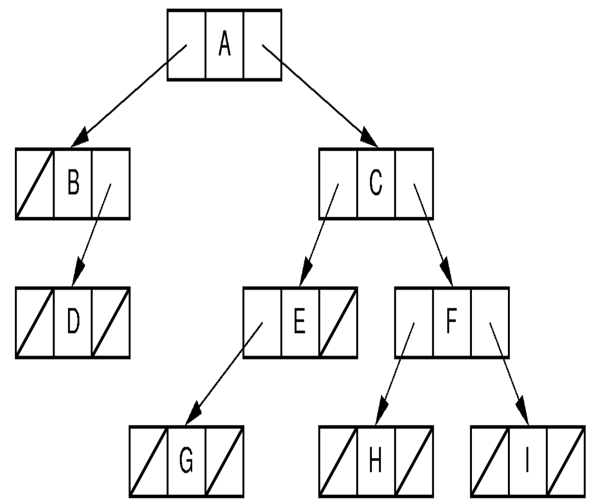


- Internal nodes store operators
  - could store a small code identifying the operator (a single byte for the operator's symbol)
- the leaves store operands
  - i.e., variable names or numbers, (considerably larger in order to handle the wider range of possible values)
  - No child pointers though

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

Space Analysis for Binary Tree Implementation

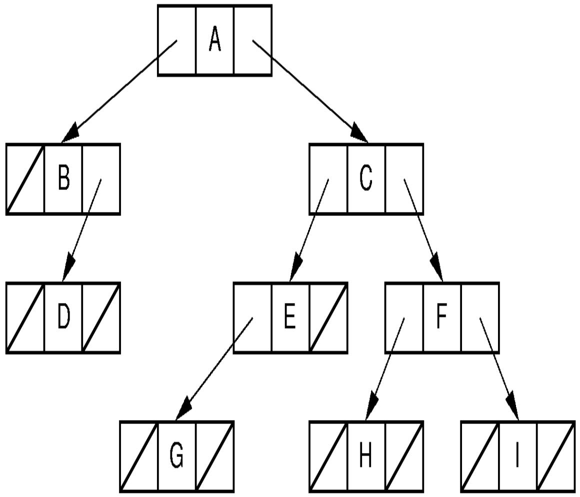
```
struct BTnode {
    int data;                        // D bytes
    struct BTnode *left, *right;     //P bytes for each one
}
```



- Every node has two pointers to its children
  - *P*: space required by a pointer
  - *D*: amount of space required by a data value

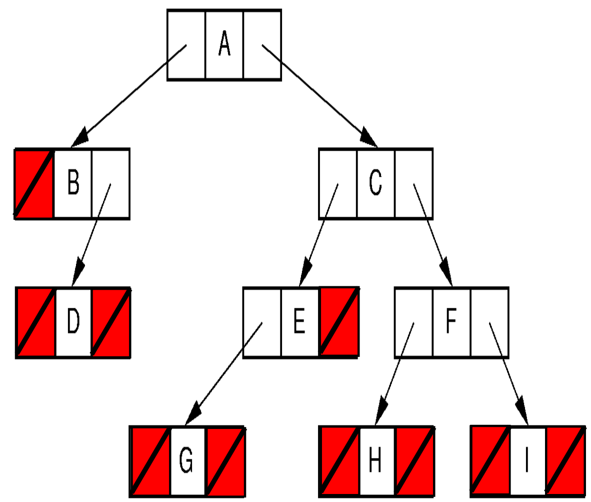
Space Analysis for Binary Tree Implementation

- 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

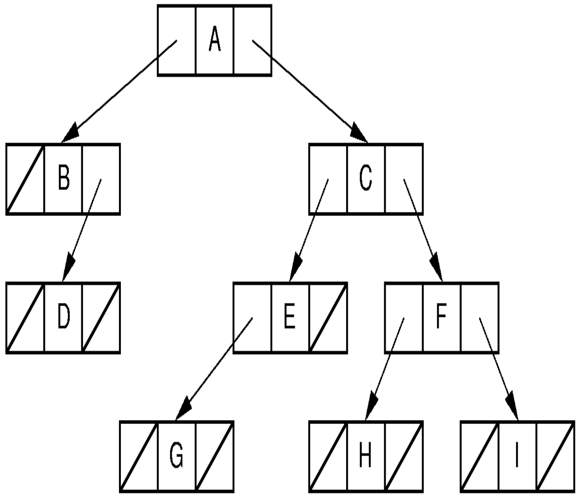
- $P = D \Rightarrow 2/3^{rd}$ of its total space is overhead
- From the Full Binary Tree Theorem: Half of the pointers are **null**.
  - half of the pointers are "wasted" **NULL values that serve only to indicate tree** structure, but which do not provide access to new data.



Space Analysis for Binary Tree Implementation

- A common implementation is not to store any actual data in a node
  - but rather a pointer to the data record.

A … B: all are pointers to data record

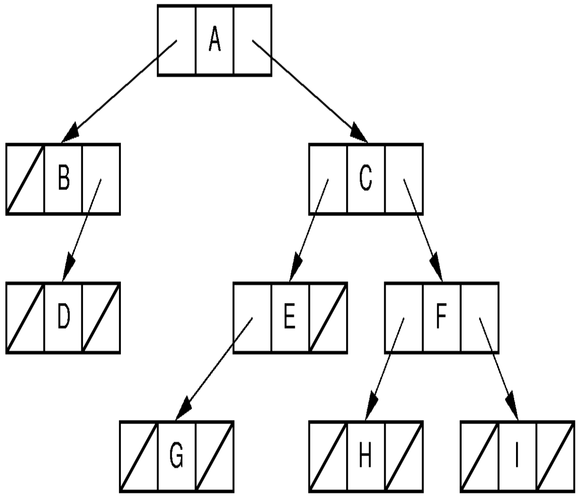


Space Analysis for Binary Tree Implementation

- A common implementation is not to store any actual data in a node
  - but rather a pointer to the data record.

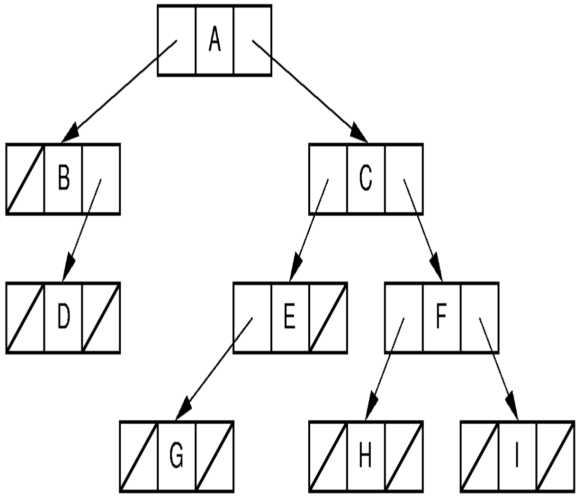
A … B: all are pointers to data record

Address	Data Records
A	Data record 1
B	Data record 2
C	Data record 3
D	Data record 4
E	Data record 5
F	Data record 6



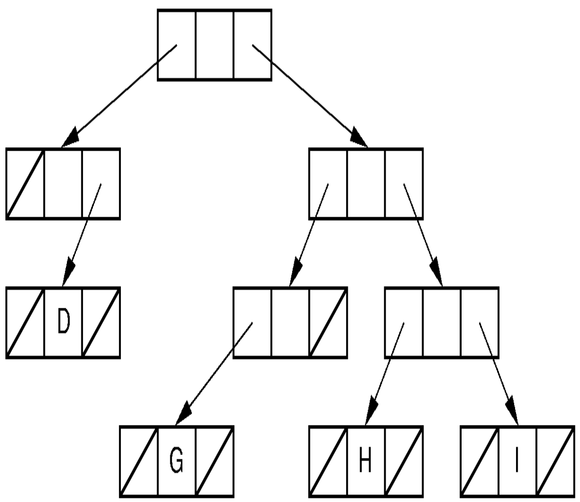
Space Analysis for Binary Tree Implementation

- In this case, each node will typically store three pointers all of which are overhead:
  - overhead fraction of $3nP/(3nP + nD) = 3P/(3P + D)$
  - $P = D$ => $3/4^{th}$ of its total space is overhead



Space Analysis for Binary Tree Implementation

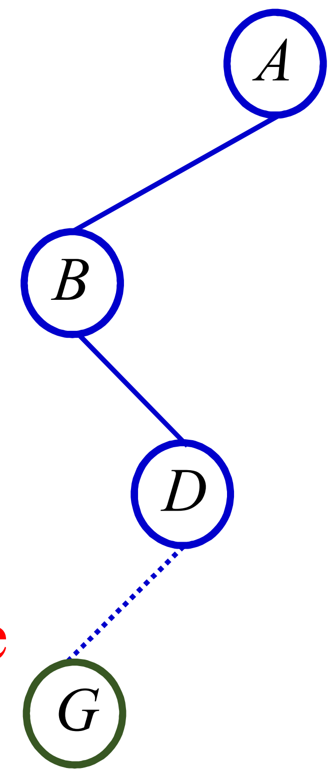
- If only leaves store data values, then the fraction of total space devoted to overhead depends on whether the tree is full.



Space Analysis for Binary Tree Implementation

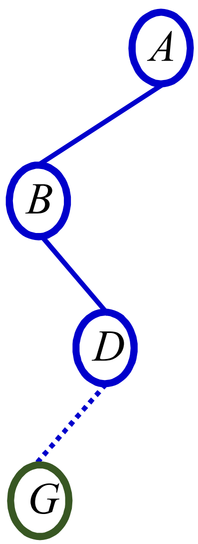
- If the tree is NOT full, then conceivably there might only be one leaf node at the end of a series of internal nodes.
  - Thus, the overhead can be an arbitrarily high percentage

Tree with ONLY one leaf node

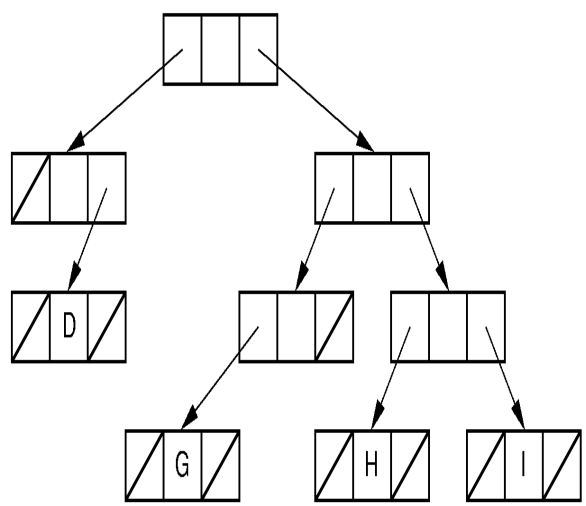


Space Analysis for Binary Tree Implementation

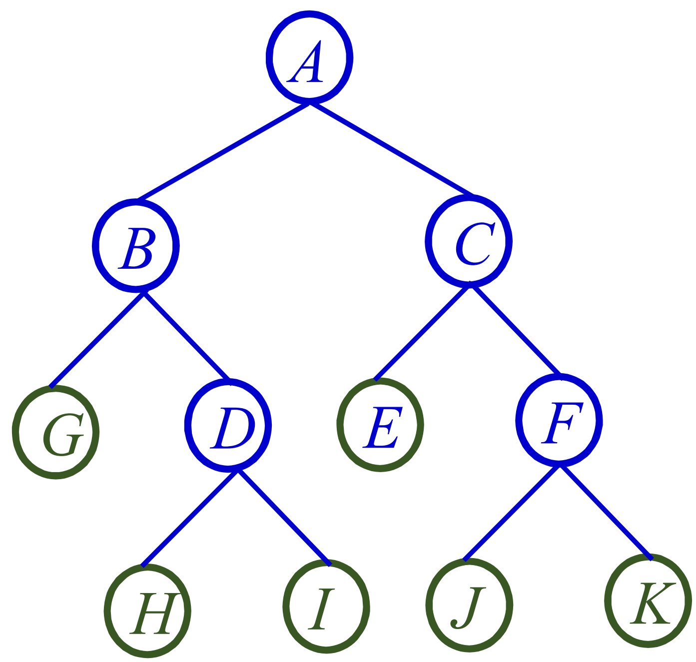
- The overhead fraction drops as the tree becomes closer to full, being lowest when the tree is truly full.
  - In this case, about one half of the nodes are internal.



Highest Overhead



Moderate Overhead



Lowest Overhead

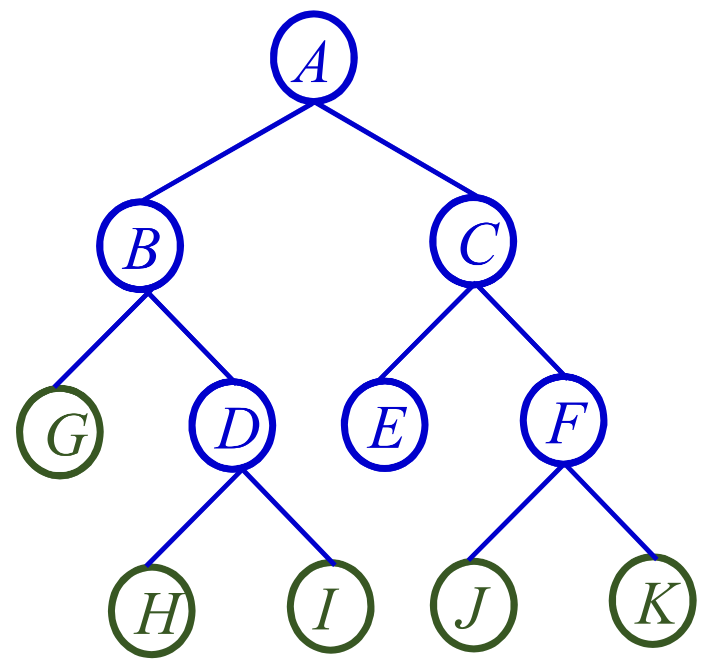
Space Analysis for Binary Tree Implementation

Eliminate pointers from the leaf nodes, but all nodes store data

$$\frac{n/2(2P)}{n/2(2P) + Dn} = \frac{P}{P + D}$$

This is 1/2 if $P = D$.

$n/2$ IN has $2P$
0 P in L
$n/2$ IN has D
$\sim n/2$ L has D



Full Binary Tree

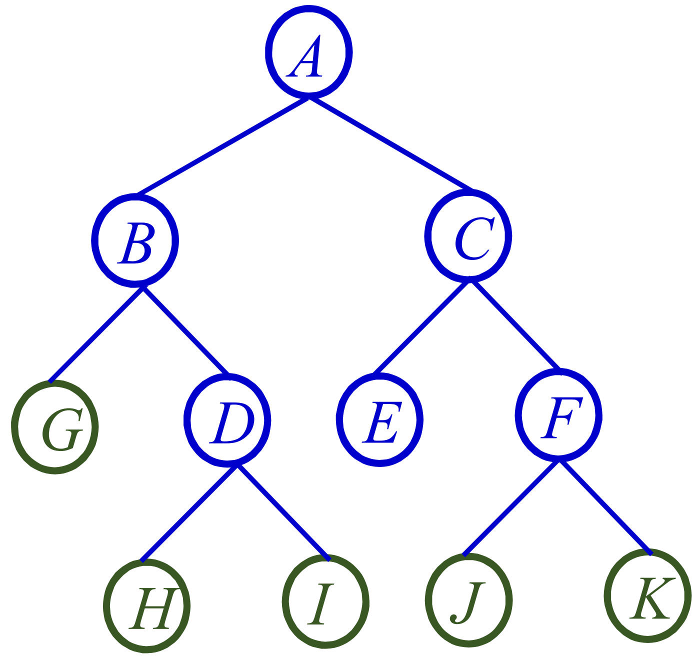
Space Analysis for Binary Tree Implementation

If data only at leaves with pointers eliminated from leaves

$(2Pn/2)/(2Pn/2 + Dn/2) = (2P)/(2P + D)$

$\Rightarrow$ 2/3 overhead (Assuming P=D).

$n/2$ IN has $2P$
$\sim n/2$ L has D



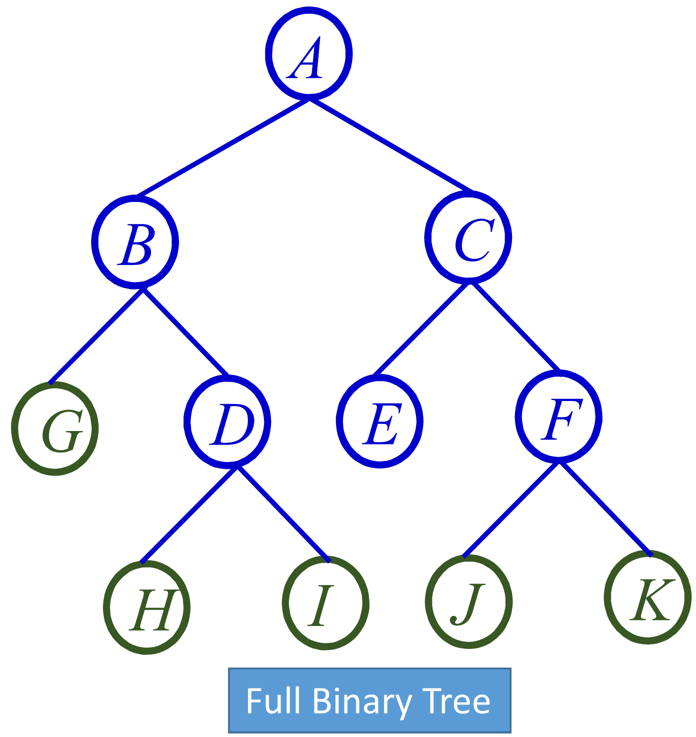
Full Binary Tree

Space Analysis for Binary Tree Implementation

A better implementation:

- internal nodes : two pointers and no data field
- leaf nodes : only a pointer to the data field

Overhead = $(3Pn/2)/(3Pn/2 + Dn/2)$
$= (3P)/(3P + D)$
=> ¾ when D = P.



$n/2$ IN has 2P
$\sim n/2$ L has $1P$
$\sim n/2$ separate data records X D