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

Instructor

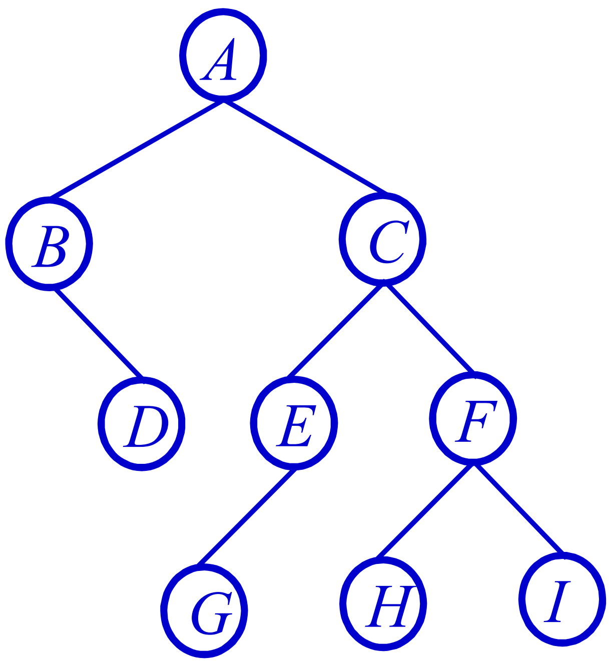
Dr Md Monirul Islam

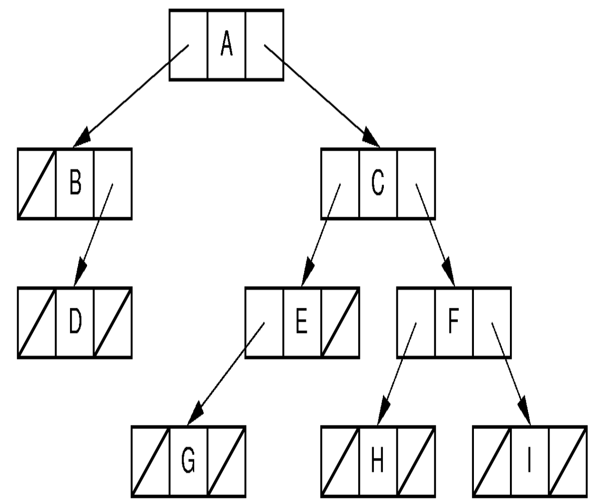
# Graphs and Trees: Representation and Search

# Binary Tree Implementation Issues

Review

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

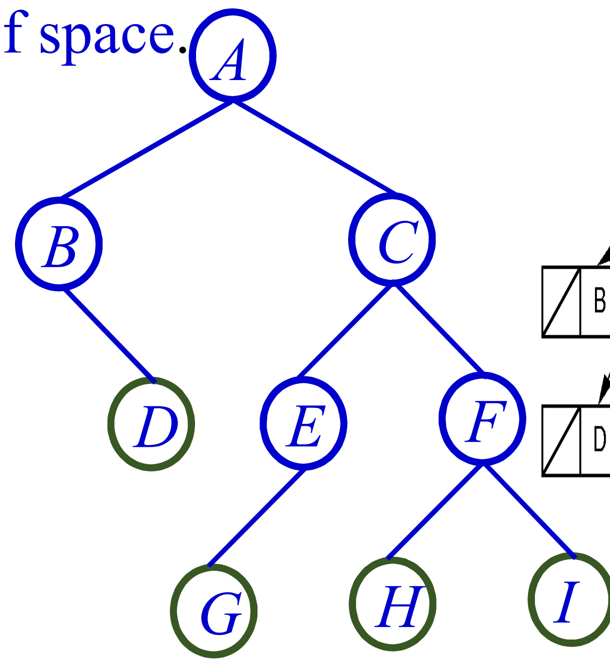


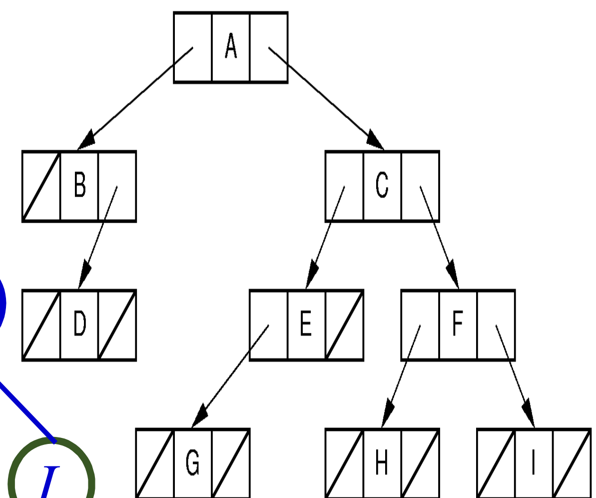



# Binary Tree Implementation Issues

Review

- 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.

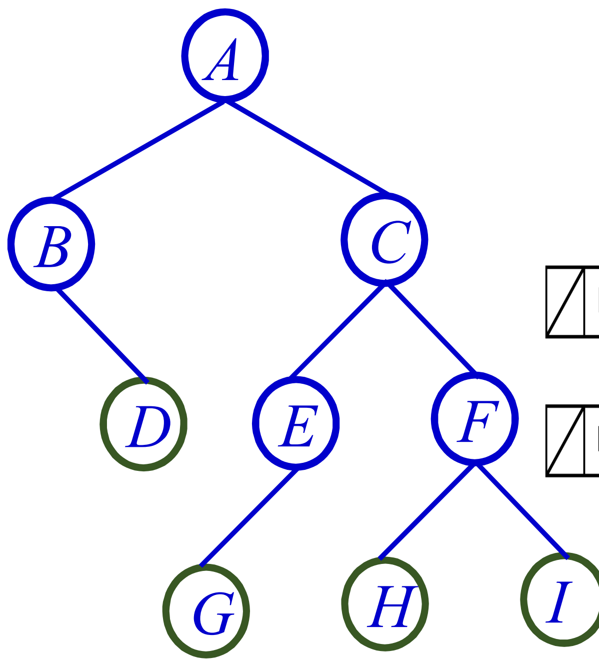


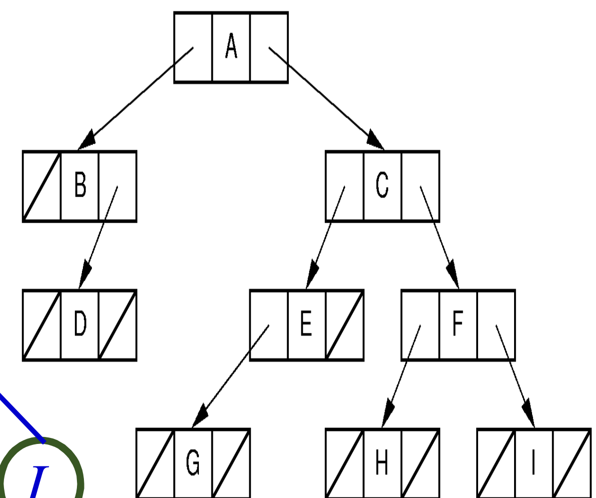



# Binary Tree Implementation Issues

Review

- 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.

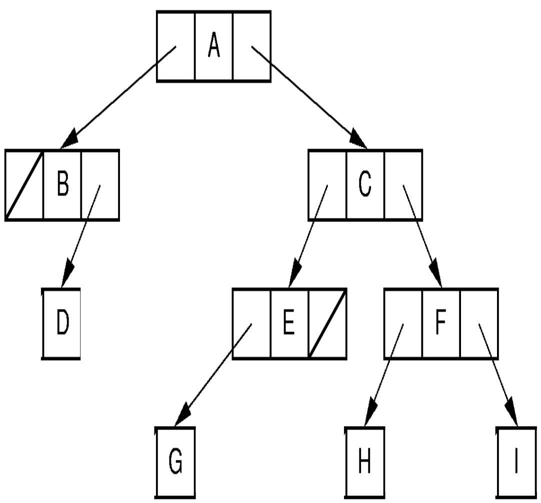


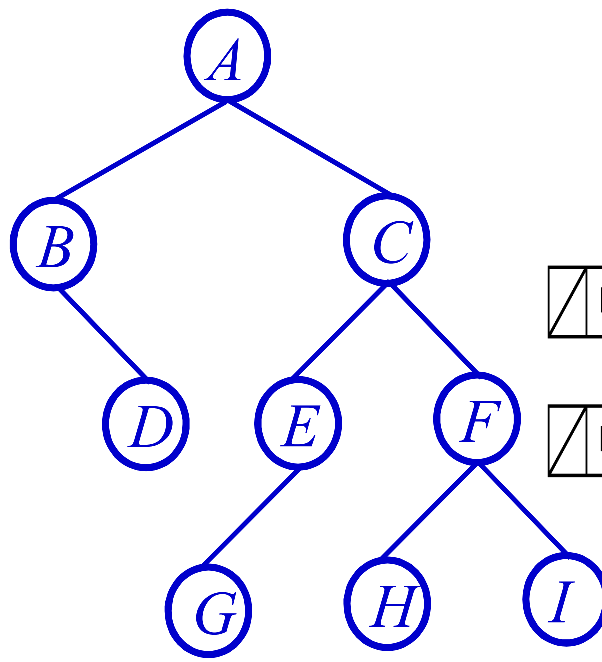
# Binary Tree Implementation Issues

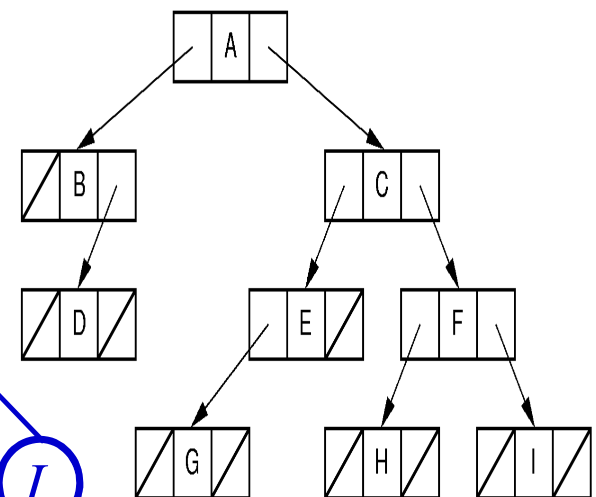
Review

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



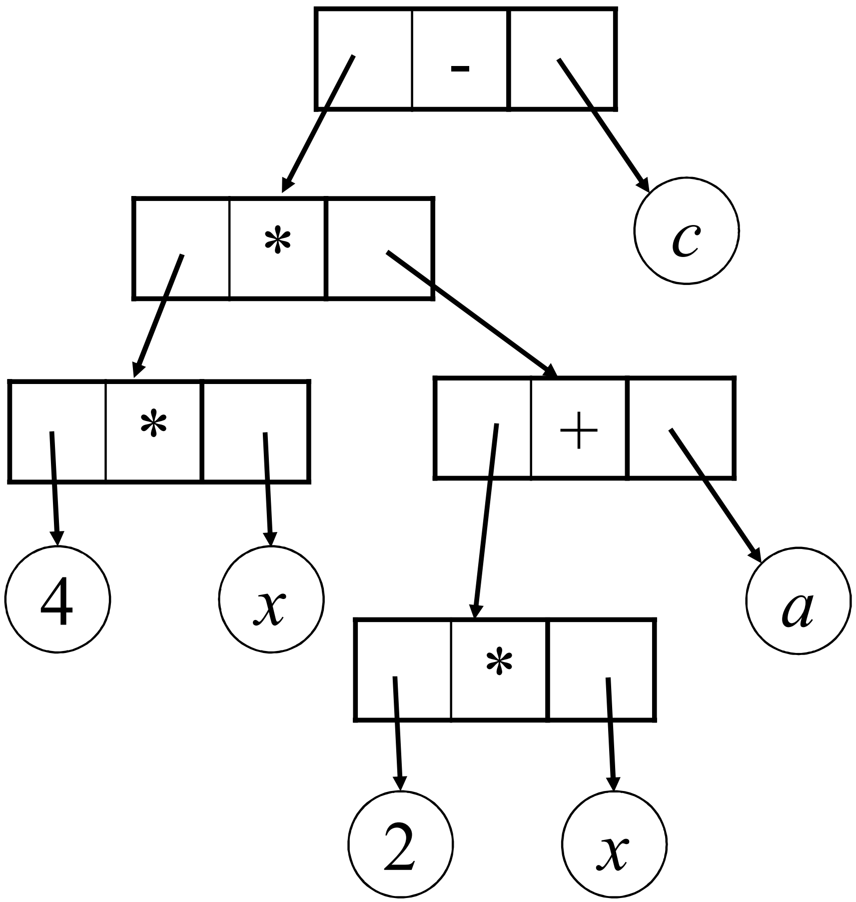

NO child pointer in leaves



# Binary Tree Implementation Issues

Review

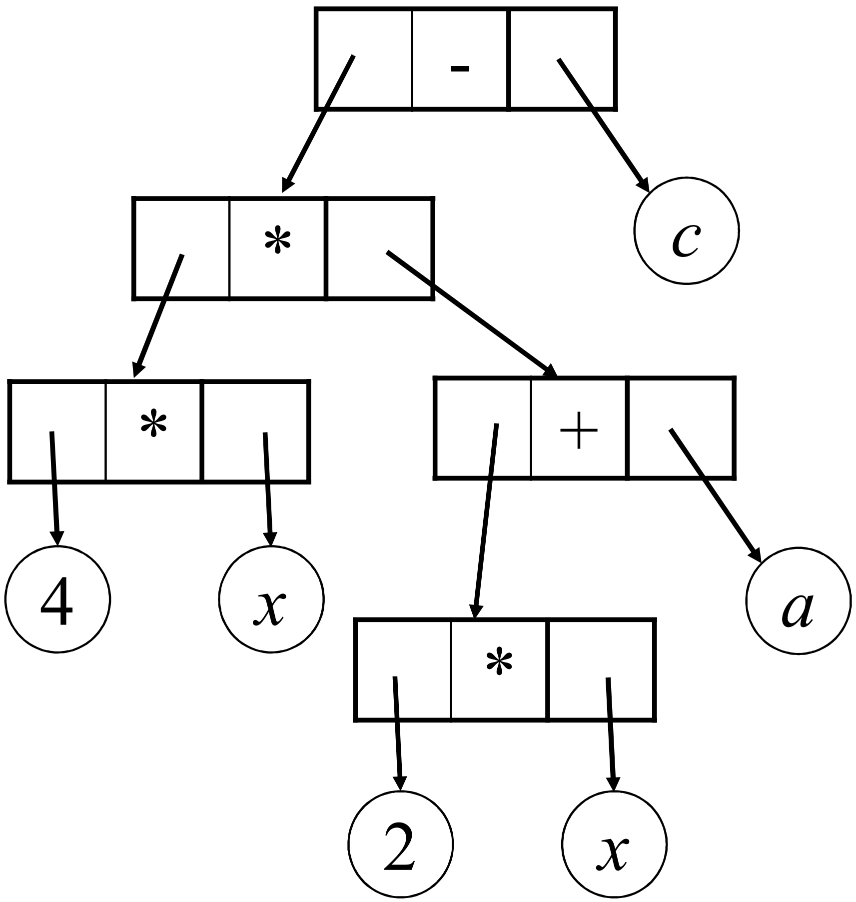



$4x\,(2x + a)$ - $c$

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

# Binary Tree Implementation Issues

Review

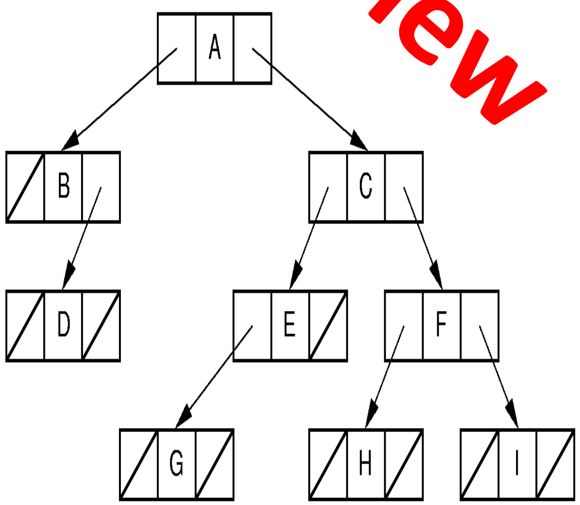



- Internal nodes store operators
  - could store a small code identifying the operator (a single byte for the operator's symbol)
- the leaves store operands
  - i.e., variable names or numbers, (considerably larger in order to handle the wider range of possible values)
  - No child pointers though

# Space Analysis for Binary Tree Implementation

Review

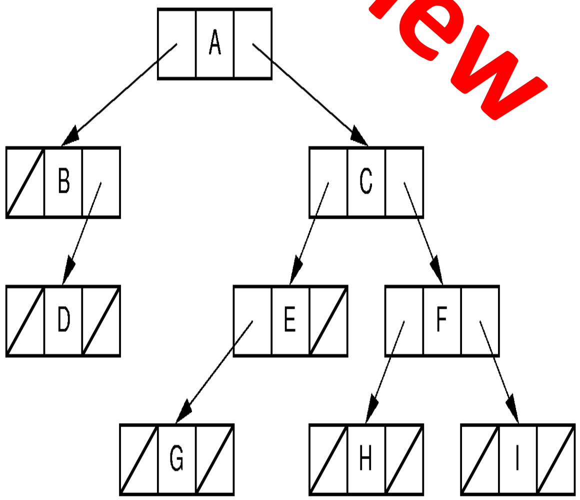
- Every node has two pointers to its children
- total space = $n(2P + D)$ for a tree of $n$ nodes
  - $P$: space required by a pointer
  - $D$: amount of space required by a data value
- So, total overhead: $2Pn$
- Overhead fraction: $2P/(2P+D)$
- $P = D$ => $2/3^{rd}$ of its total space is overhead



# Space Analysis for Binary Tree Implementation

Review

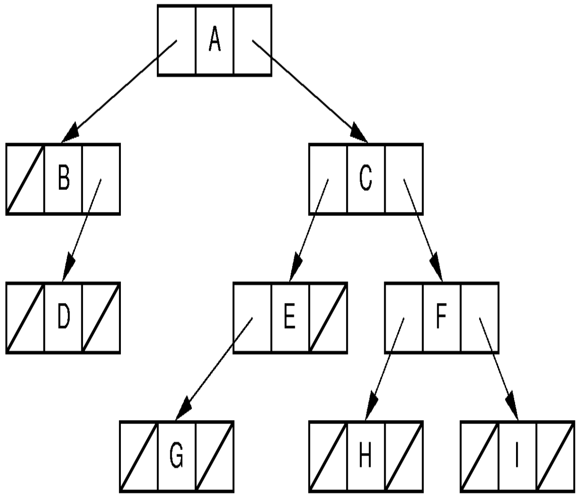
- $P = D$ => 2/3rd of its total space is overhead
- From the Full Binary Tree Theorem: Half of the pointers are **null**.
  - half of the pointers are "wasted" **NULL values that serve only to indicate tree structure**, but which do not provide access to new data.



# 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

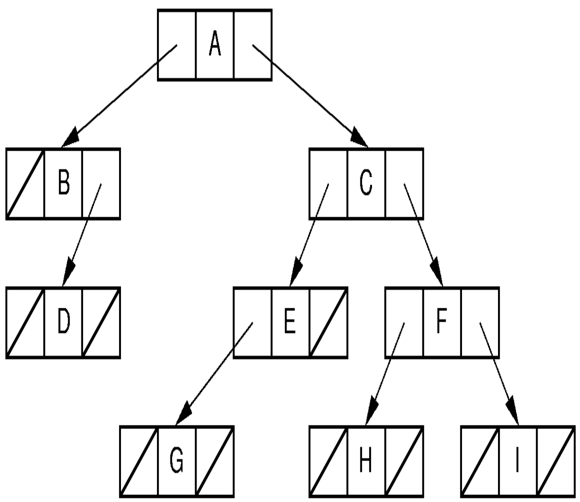


# Space Analysis for Binary Tree Implementation

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

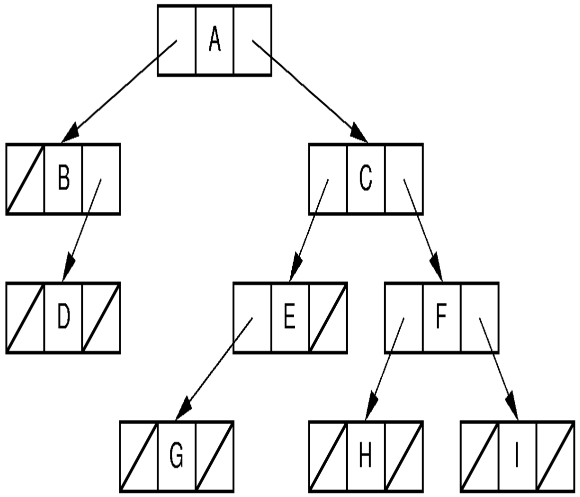
A … B: all are pointers to data record

| 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 |



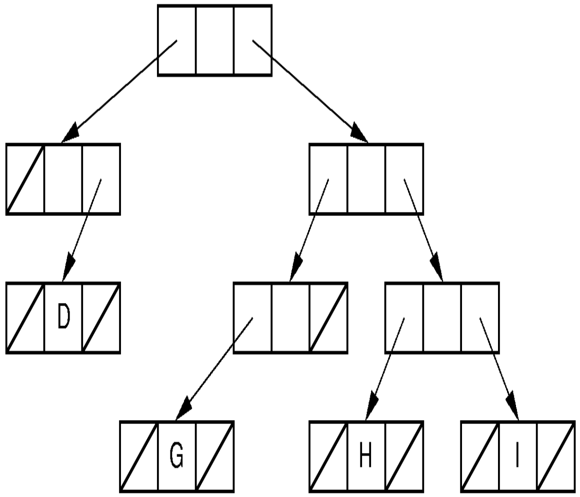
# Space Analysis for Binary Tree Implementation

- 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



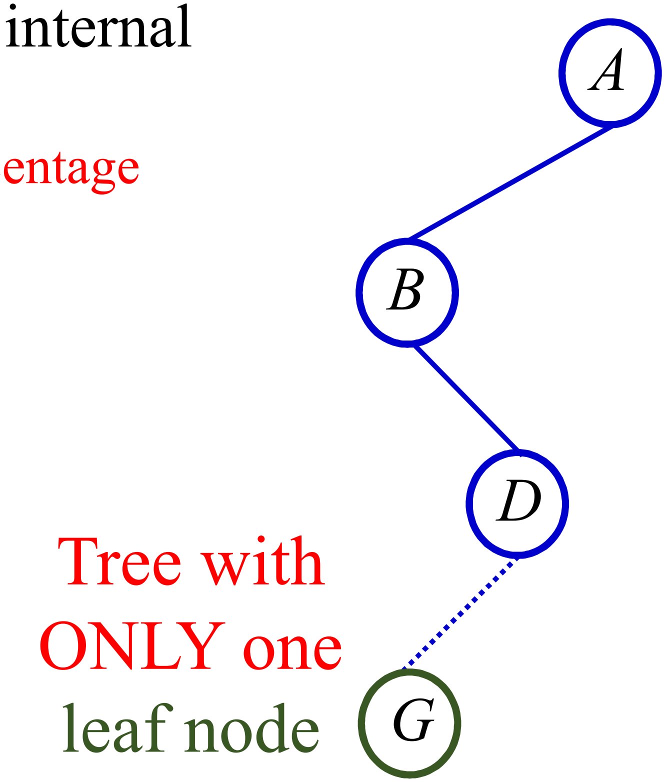
# Space Analysis for Binary Tree Implementation

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



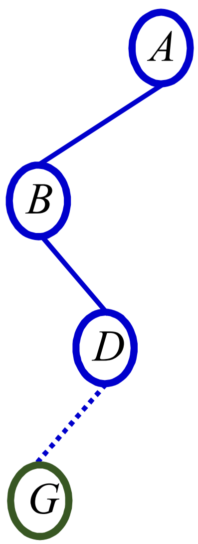
# Space Analysis for Binary Tree Implementation

- 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

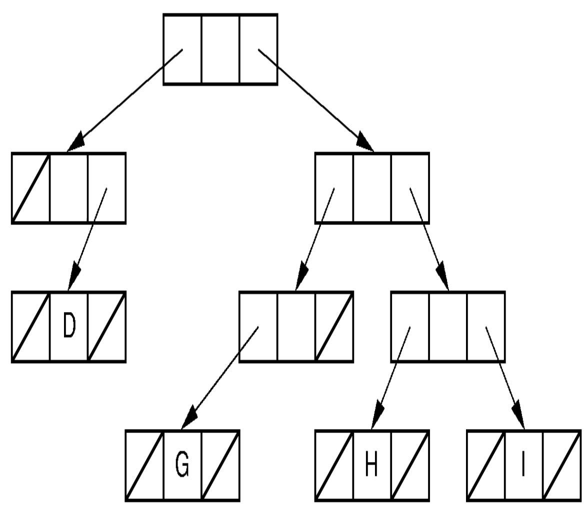


# Space Analysis for Binary Tree Implementation

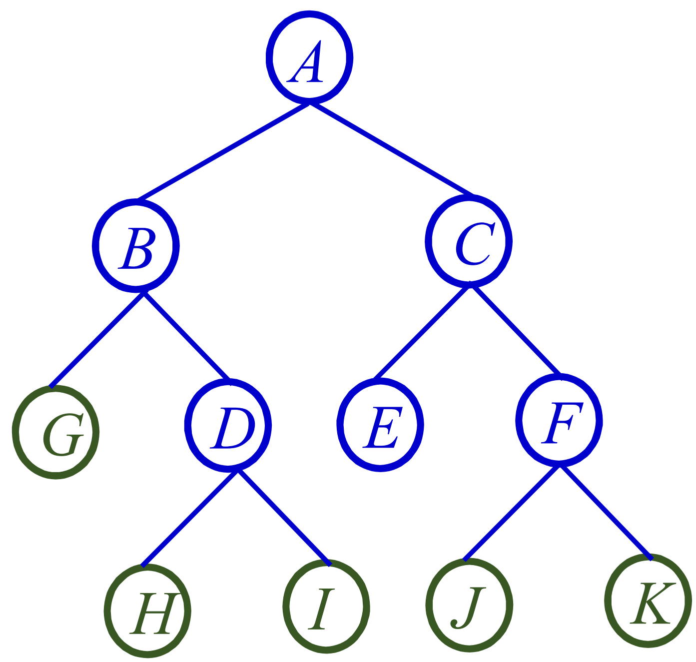
- The overhead fraction drops as the tree becomes closer to full, being lowest when the tree is truly full.
  - In this case, about one half of the nodes are internal.




Highest Overhead




Moderate Overhead




Lowest Overhead

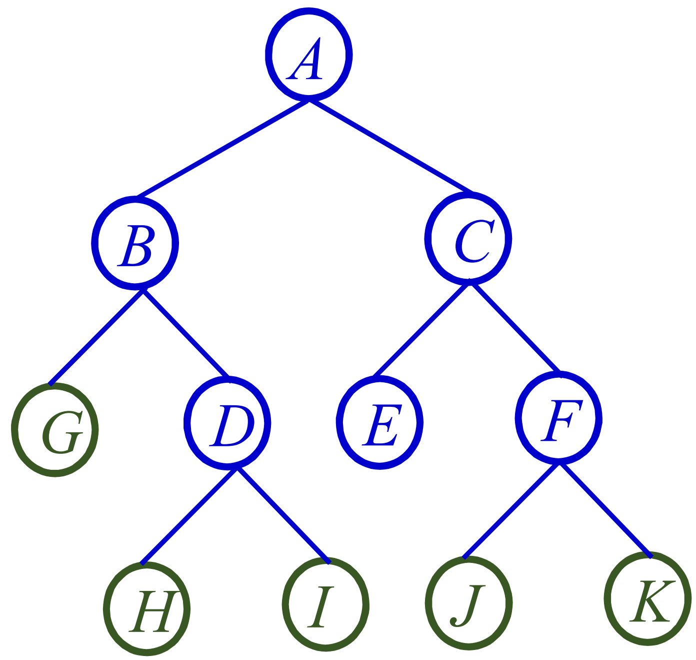
# Space Analysis for Binary Tree Implementation

Eliminate pointers from the leaf nodes:

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

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

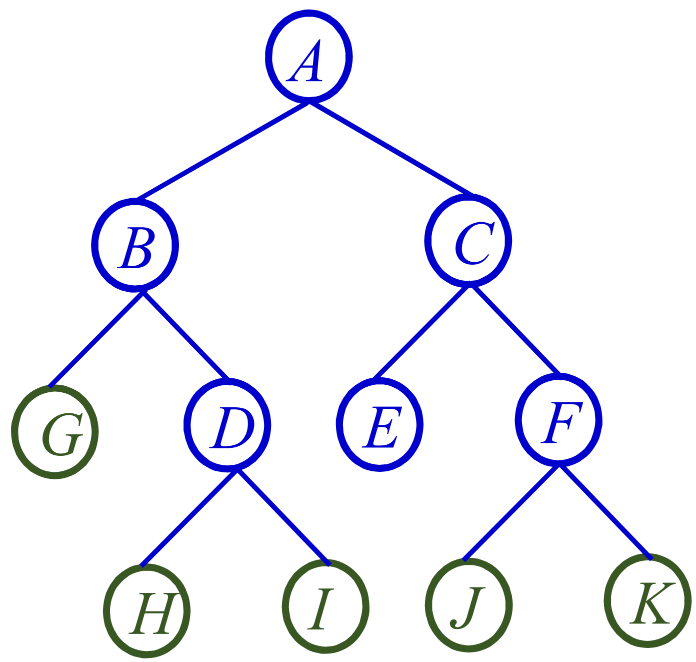
# Space Analysis for Binary Tree Implementation

If data only at leaves with pointers eliminated

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

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

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




Full Binary Tree

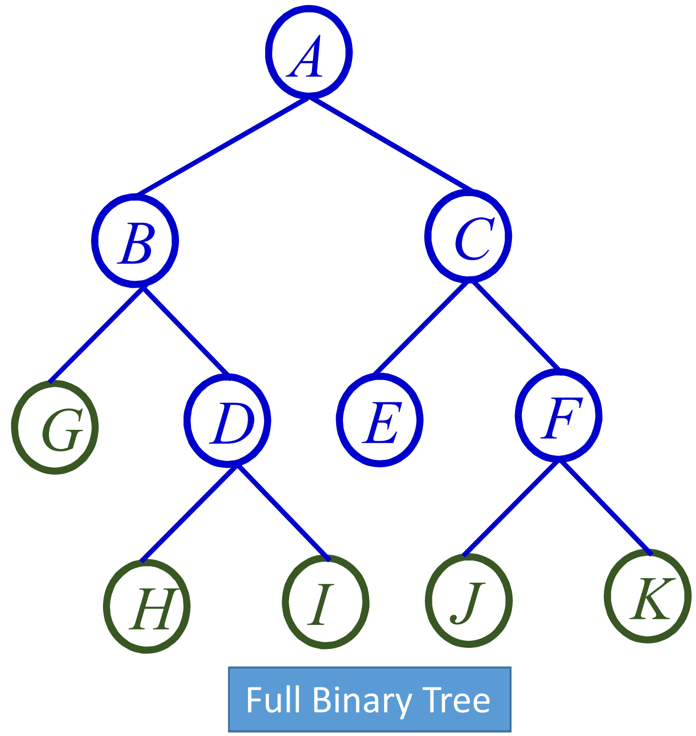
# Space Analysis for Binary Tree Implementation

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 $D$



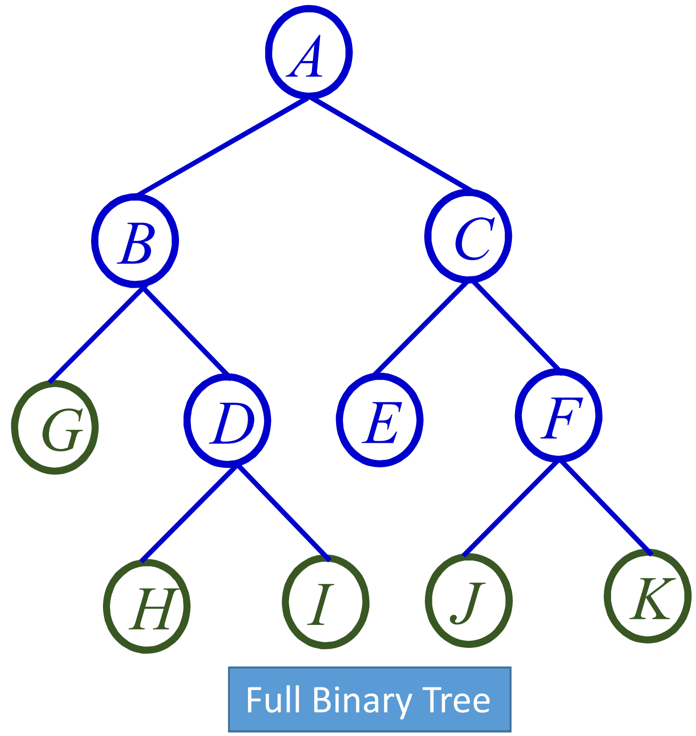
# Space Analysis for Binary Tree Implementation

A better implementation:

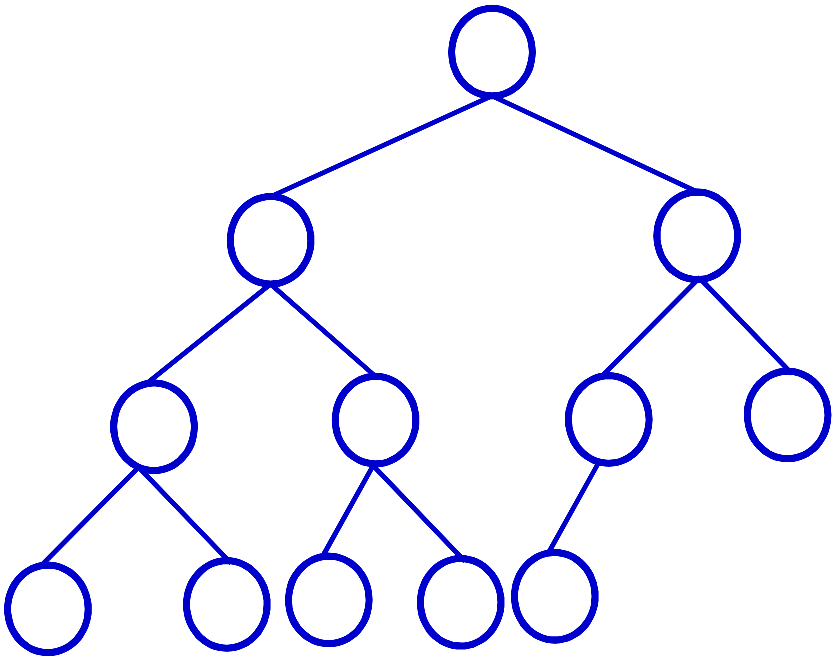
- internal nodes : two pointers and no data field
- leaf nodes : only a pointer to the data field

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 $D$

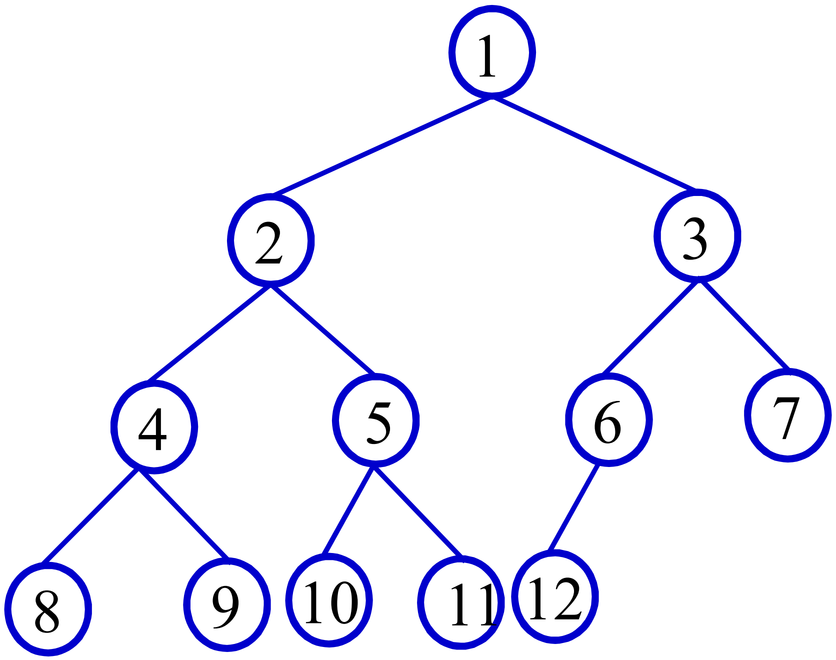


# Binary Tree Implementation:
## Complete Binary Tree



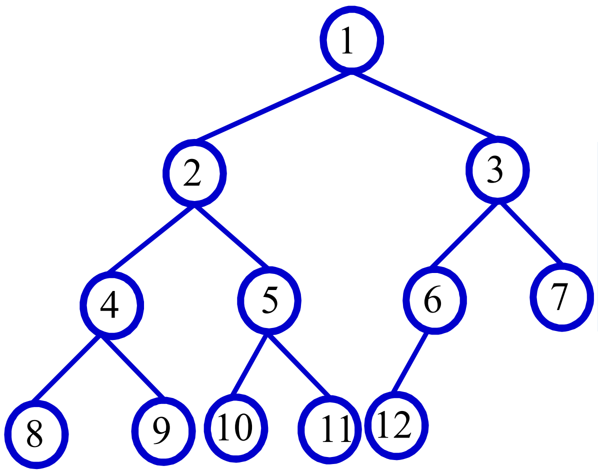
Complete Binary Tree

# Binary Tree Implementation: Complete Binary Tree



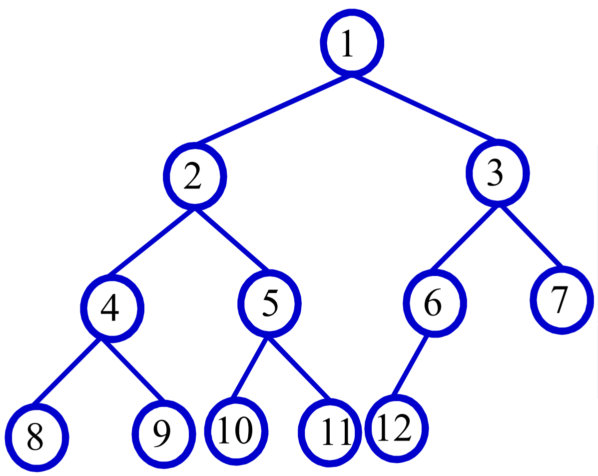

Complete Binary Tree

# Binary Tree Implementation:
## Complete Binary Tree



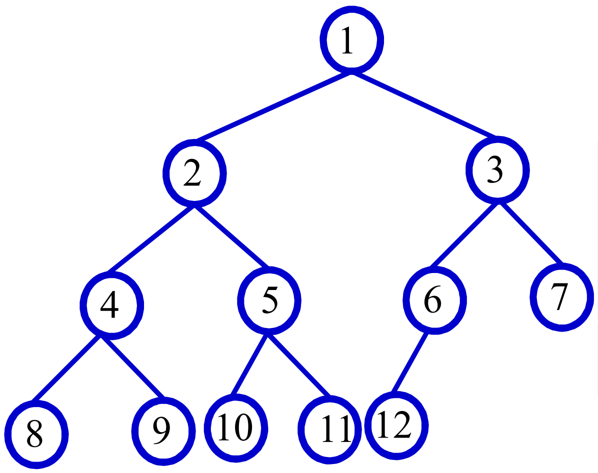

| Node Position in Array | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

# Binary Tree Implementation:
## Complete Binary Tree




| Node Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Parent | -- | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 |

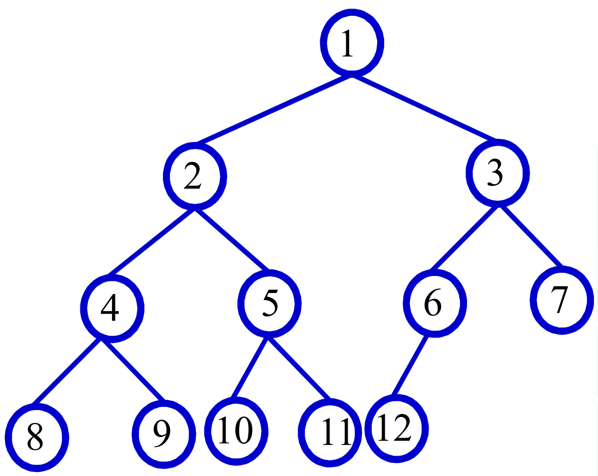
# Binary Tree Implementation:
## Complete Binary Tree




| Node Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Parent | -- | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 |

parent($i$) = floor ($i$/2);

# Binary Tree Implementation: Complete Binary Tree

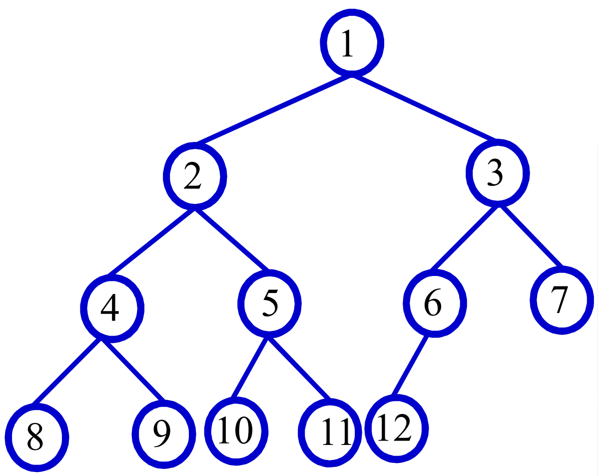



| Node Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Parent | -- | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 |
| Left Child | 2 | 4 | 6 | 8 | 10 | 12 | -- | -- | -- | -- | -- | -- |

parent($i$) = floor ($i$/2);

left($i$) = 2*$i$;

# Binary Tree Implementation: Complete Binary Tree



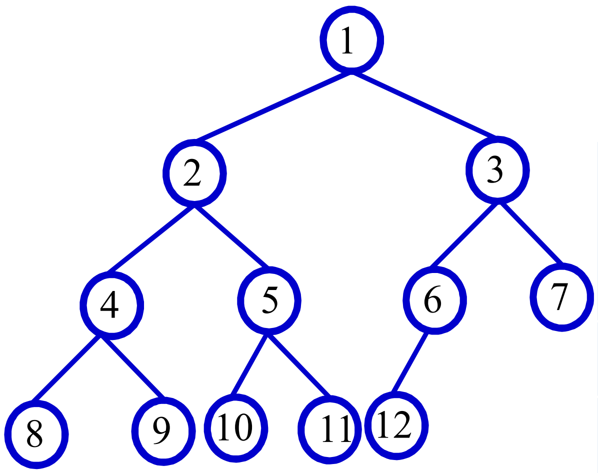

| **Node Position** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Parent | -- | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 |
| Left Child | 2 | 4 | 6 | 8 | 10 | 12 | -- | -- | -- | -- | -- | -- |
| **Right Child** | **3** | **5** | **7** | **9** | **11** | **--** | **--** | **--** | **--** | **--** | **--** | **--** |

parent($i$) = floor ($i$/2);

left($i$) = 2*$i$;

right($i$) = 2*$i$ +1;

# Binary Tree Implementation:
## Complete Binary Tree




| Node Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Parent | -- | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 |
| Left Child | 2 | 4 | 6 | 8 | 10 | 12 | -- | -- | -- | -- | -- | -- |
| Right Child | 3 | 5 | 7 | 9 | 11 | -- | -- | -- | -- | -- | -- | -- |
| **Left Sibling** | **--** | **--** | **2** | **--** | **4** | **--** | **6** | **--** | **8** | **--** | **10** | **--** |

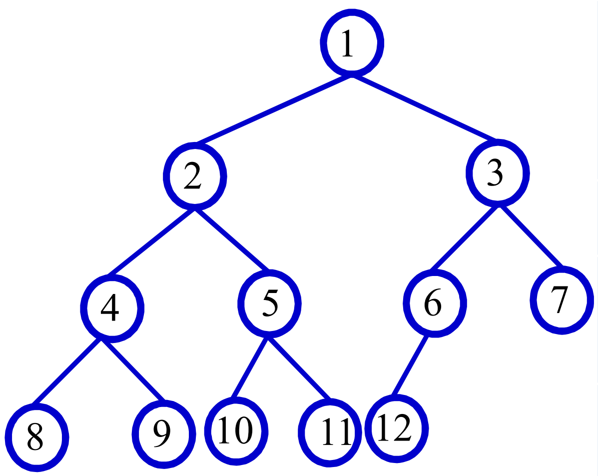
parent($i$) = floor ($i$/2);

left($i$) = 2*$i$;

right($i$) = 2*$i$ +1;

leftSibling($i$) = $i$-1, if $i$ is odd;

# Binary Tree Implementation: Complete Binary Tree




| **Node Position** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Parent | -- | 1 | 1 | 2 | 2 | 3 | 3 | 4 | 4 | 5 | 5 | 6 |
| Left Child | 2 | 4 | 6 | 8 | 10 | 12 | -- | -- | -- | -- | -- | -- |
| Right Child | 3 | 5 | 7 | 9 | 11 | -- | -- | -- | -- | -- | -- | -- |
| Left Sibling | -- | -- | 2 | -- | 4 | -- | 6 | -- | 8 | -- | 10 | -- |
| **Right Sibling** | **--** | **3** | **--** | **5** | **--** | **7** | **--** | **9** | **--** | **11** | **--** | **--** |

parent($i$) = floor ($i$/2);

left($i$) = 2*$i$;

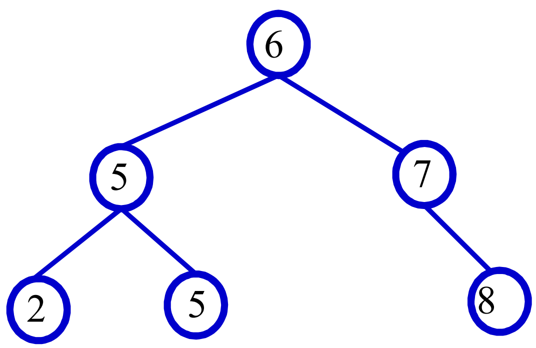
right($i$) = 2*$i$ +1;

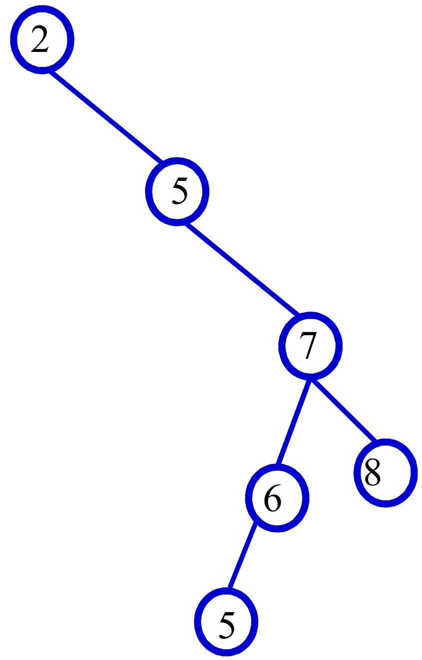
leftSibling($i$) = $i$-1, if $i$ is odd;

rightSibling($i$) = $i$+1, if $i$ is even;

# Binary Search Tree

- A Binary tree
- Three pointers in each node: left, right, parent
- Maintains a special property for each node **Binary Search Tree property**

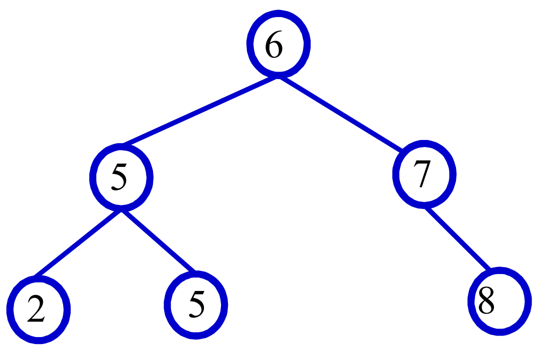


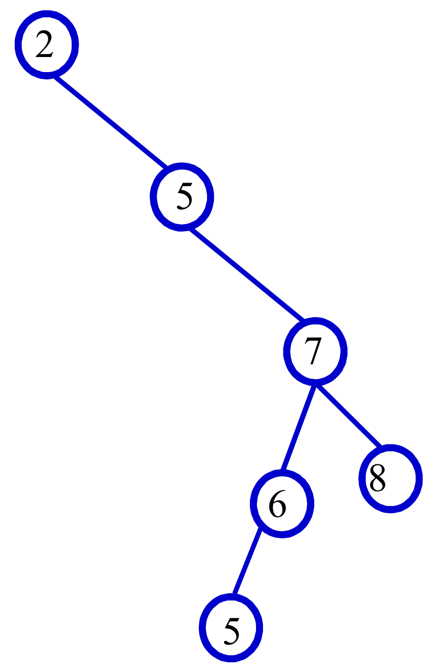



# Binary Search Tree

**BST property**

All elements stored in the left subtree of a node with value *K* have values <= *K*.
All elements stored in the right subtree of a node with value *K* have values >= *K*.



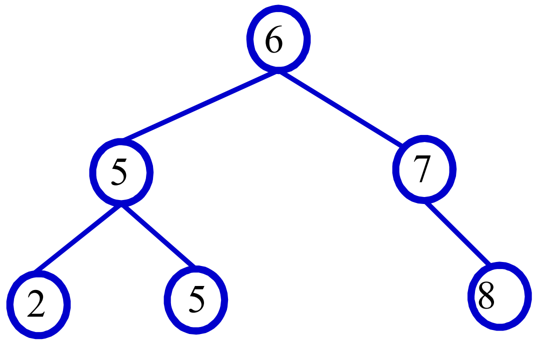
# Binary Search Tree

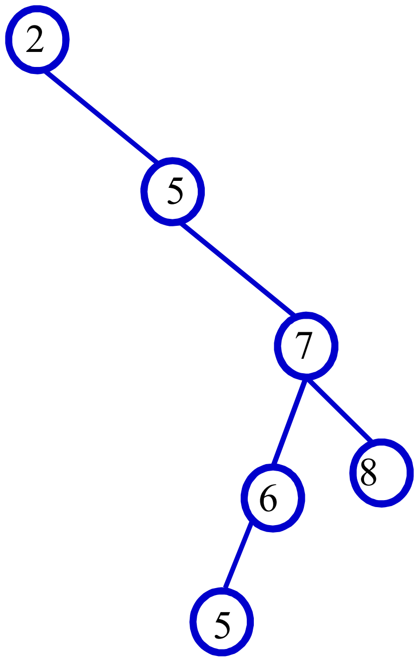
**BST property**

Let $x$ be a node in a binary search tree.

If $y$ is a node in the left subtree of $x$, then $y.key \leq x.key$

If $y$ is a node in the right subtree of $x$, then $y.key \geq x.key$.

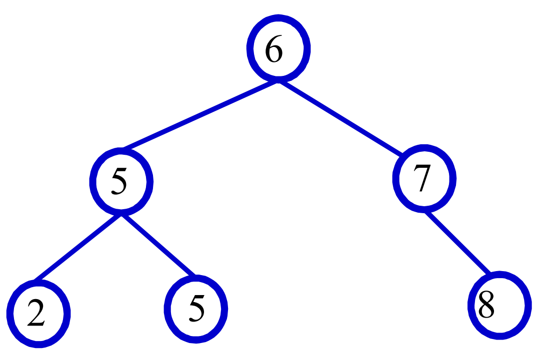


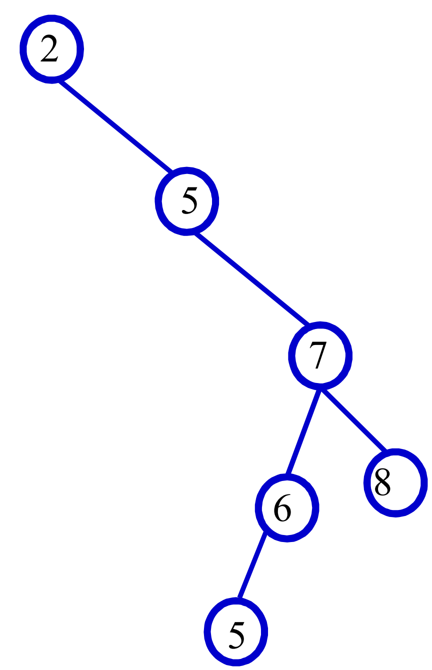



# Binary Search Tree

**Inorder traversal of a BST**

**Traversal Outcome:?**

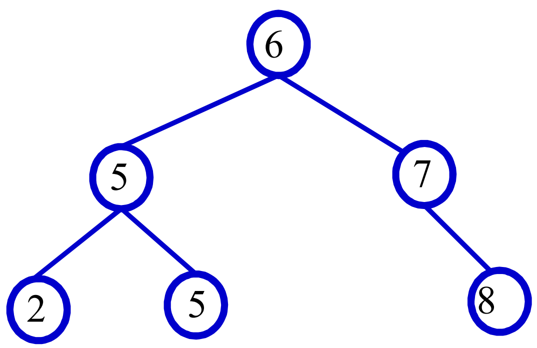


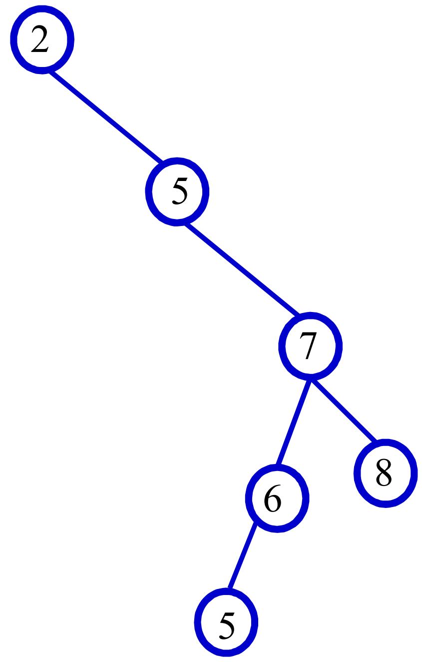



# Binary Search Tree

**Inorder traversal of a BST**

**Traversal Outcome: 2 5 5 6 7 8**



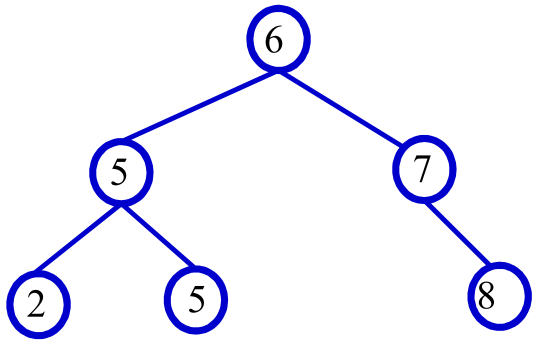



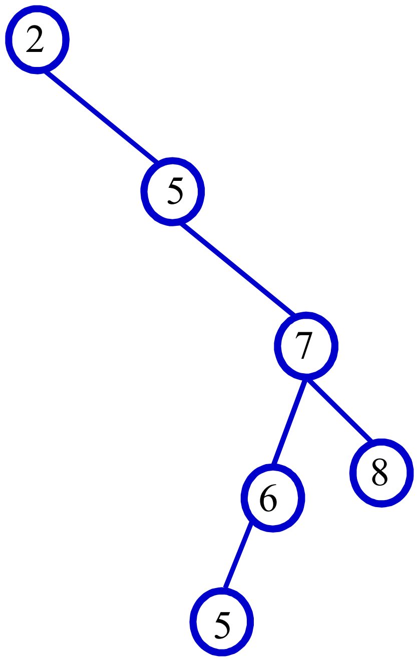
# Binary Search Tree

**Inorder traversal of a BST**

**Traversal Outcome: 2 5 5 6 7 8**

Same list of keys but different BST shape.



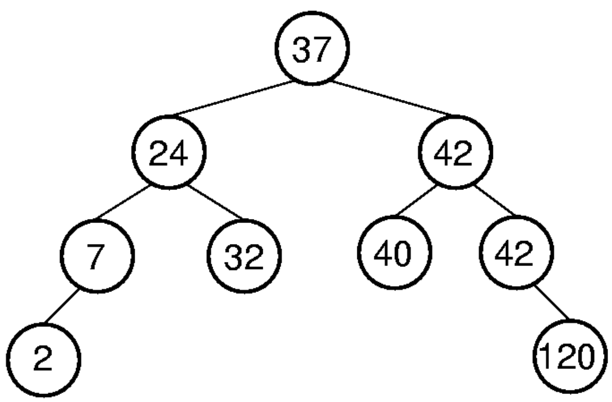



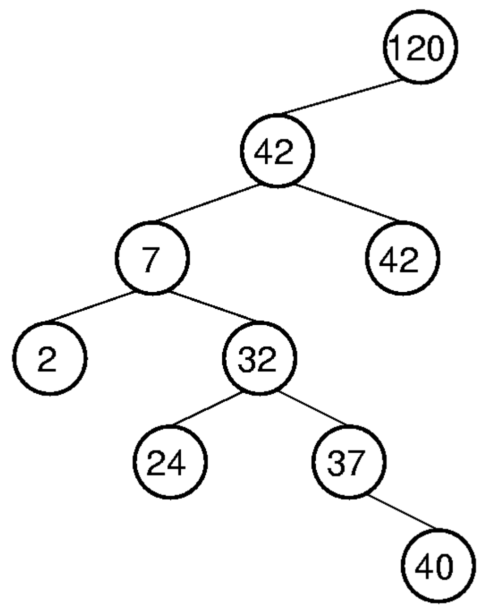
# Binary Search Tree

**Another Example**

**Traversal Outcome:** 2, 7, 24, 32, 37, 40, 42, 42, 120

We also get two different BSTs.



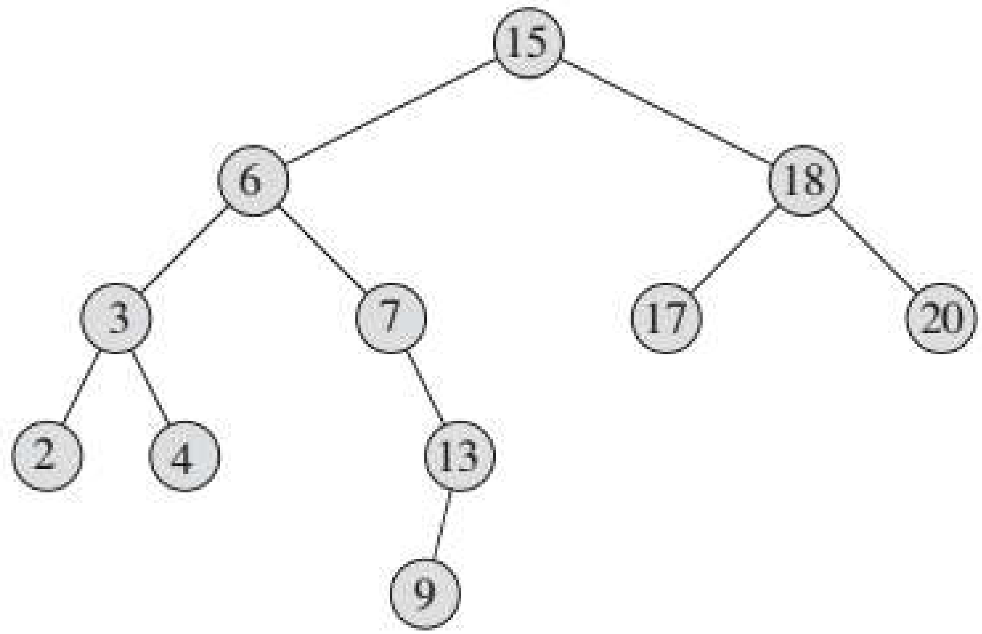



# BST Operations

- Search for a key
- Minimum
- Maximum
- Successor
- Predecessor
- Insert
- Delete

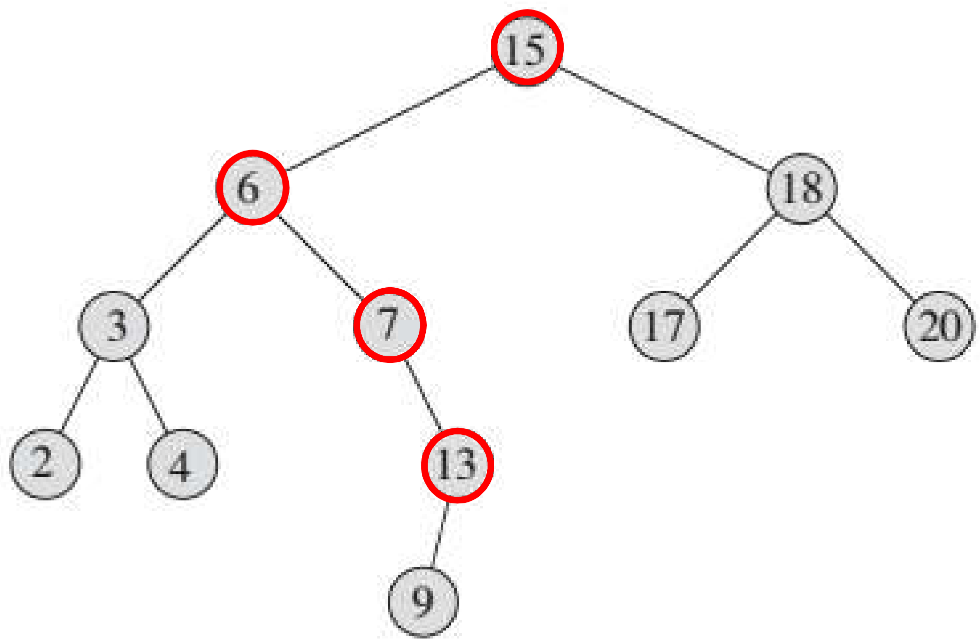
# BST Operation: Search

```
TREE_SEARCH (x, k)
if x == NULL or k == x->key
return x
if k < x->key
return TREE_SEARCH(x->left, k)
else return TREE_SEARCH(x->right, k)
```



# BST Operation: Search

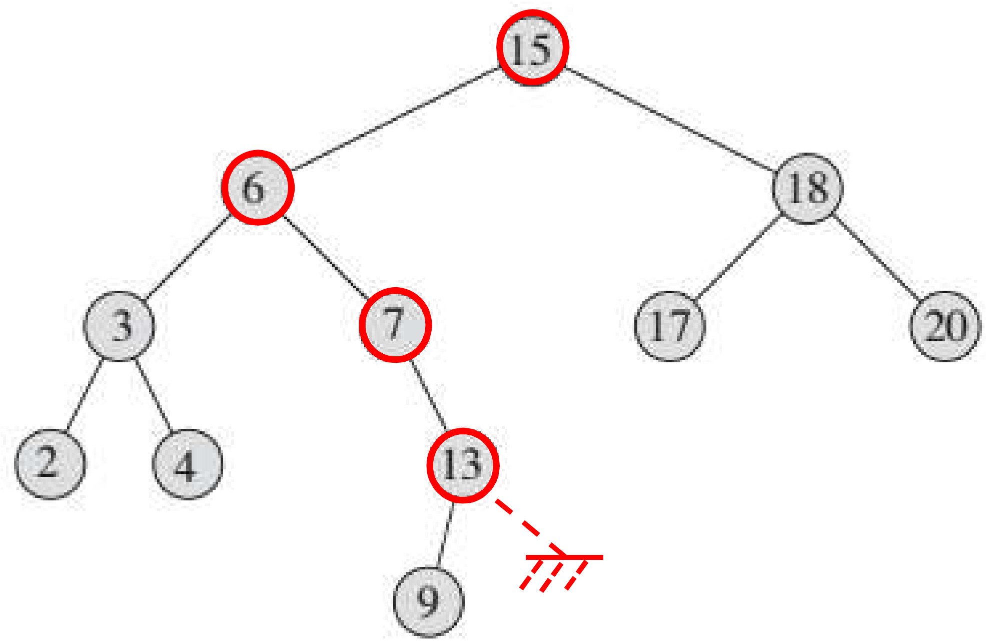
```
TREE_SEARCH (x, k)
if x == NULL or k == x->key
return x
if k < x->key
return TREE_SEARCH(x->left, k)
else return TREE_SEARCH(x->right, k)
```




**Search for 13**

# BST Operation: Search

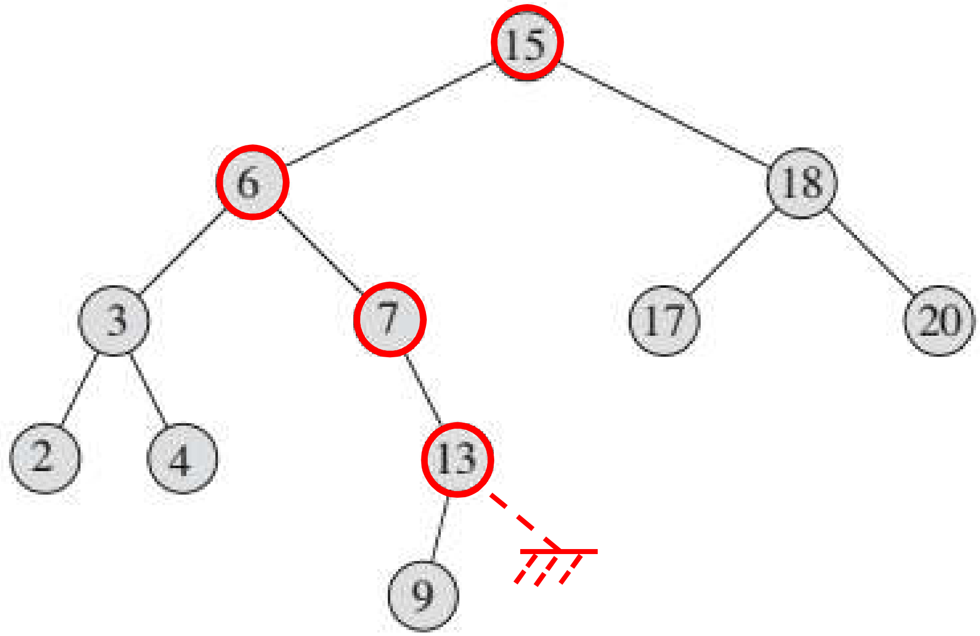
TREE_SEARCH ($x$, $k$)
1 **if** $x$ == NULL or $k$ == $x$->*key*
2 **return** $x$
3 **if** $k$ < $x$->*key*
4 **return** TREE_SEARCH($x$->*left*, $k$)
5 **else return** TREE_SEARCH($x$->*right*, $k$)




**Search for 14**

# BST Operation: Search

```
TREE_SEARCH (x, k)
if x == NULL or k == x->key
return x
if k < x->key
return TREE_SEARCH(x->left, k)
else return TREE_SEARCH(x->right, k)
```




**Complexity: O(*h*)**