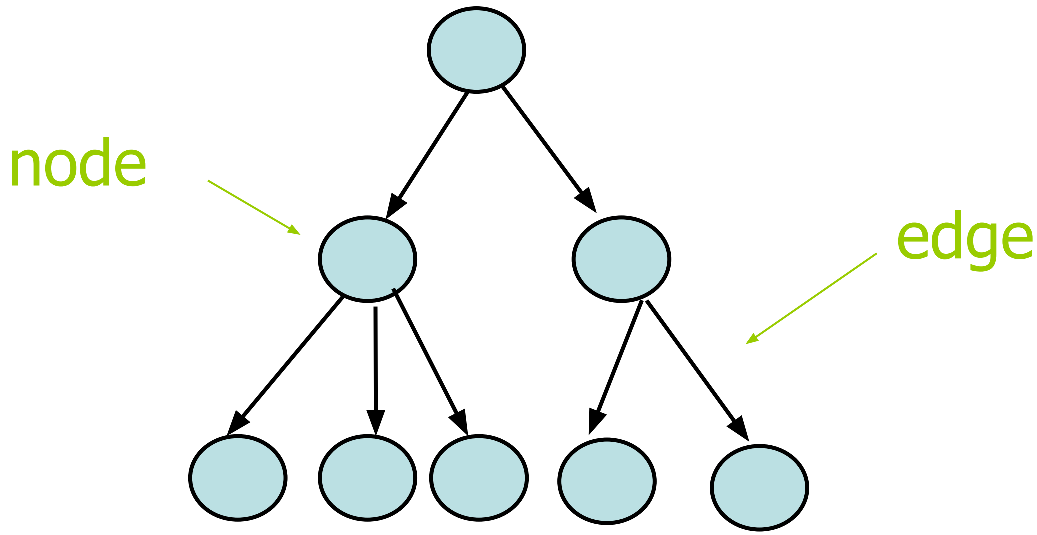
Data Structures and Algorithms

Dr. Odelu Vanga
Department of Computer Science and Engineering
Indian Institute of Information Technology Sri City

- Binary Trees & Tree Traversal
- BFS and DFS
- Spanning Trees

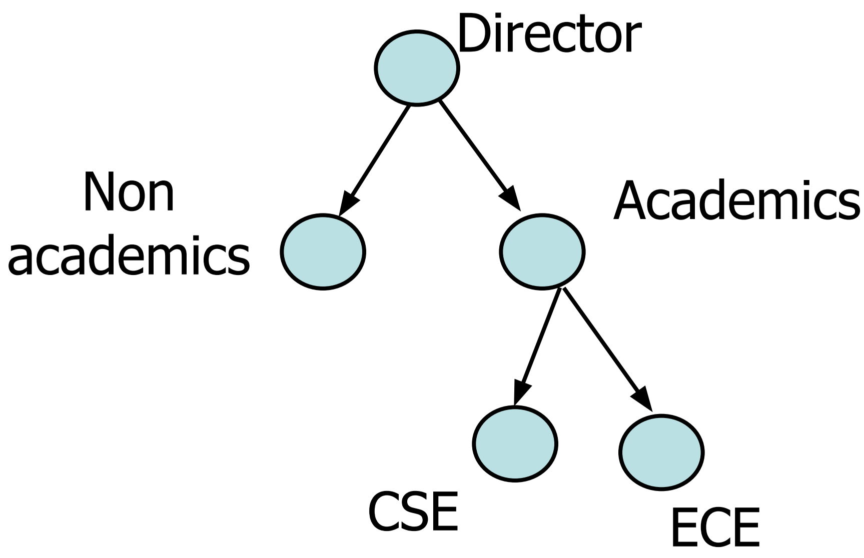
What is a tree?



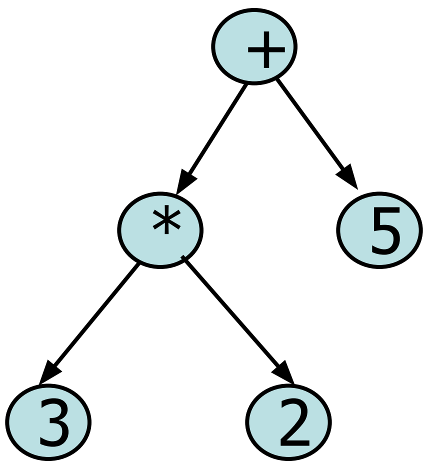
- Represent hierarchical relationship
- Consists of nodes and edges
- Node represents an object
- Edge represents relationship

Some applications of Trees

Organization Chart



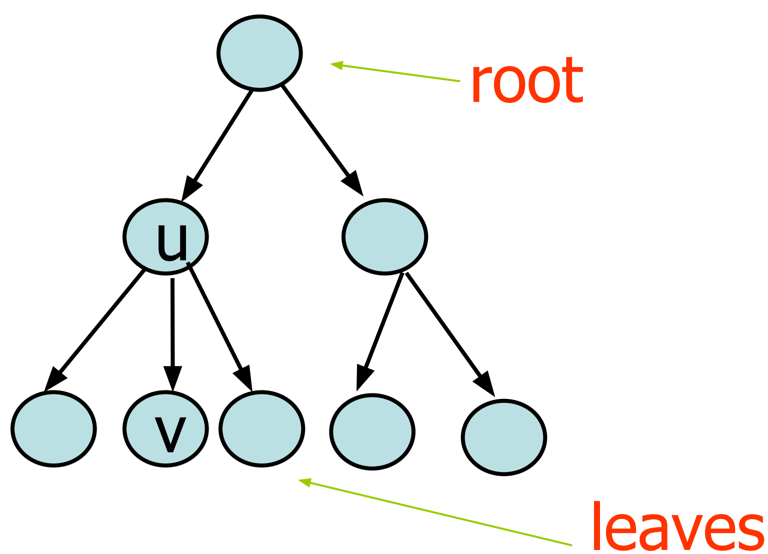
Expression Tree



Terminology I

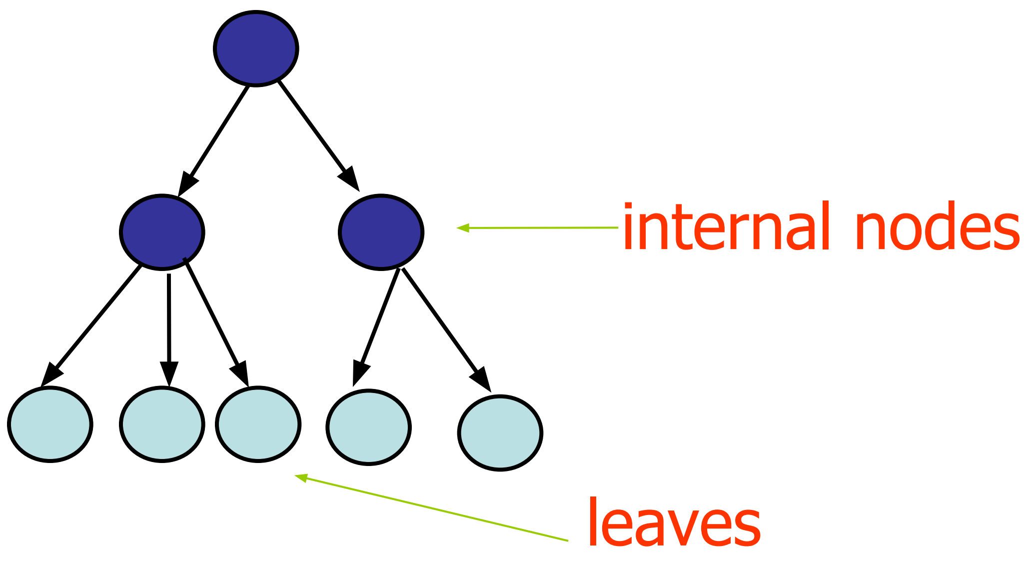
- For any two nodes u and v, if there is an edge pointing from u to v, u is called the parent of v while v is called the child of u.
- Such edge is denoted as **(u, v)**.
- Node without parent, which is called the root.
- The nodes without children are called leaves.

u: parent of v
v: child of u



Terminology II

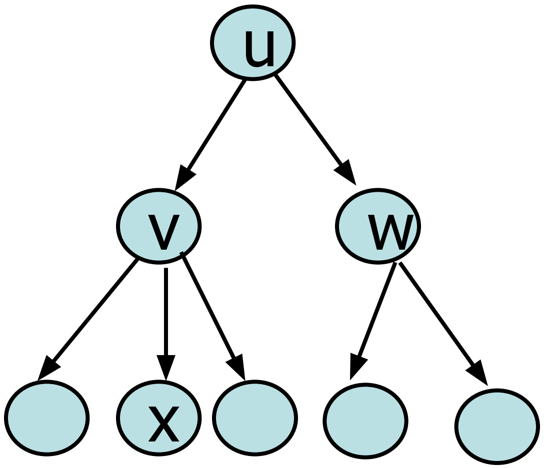
- Nodes without children are called leaves.
- Otherwise, they are called internal nodes.



Terminology III

- If two nodes have the same parent, they are siblings.
- A node u is an ancestor of v if u is parent of v or parent of parent of v or …
- A node v is a descendent of u if v is child of v or child of child of v or …

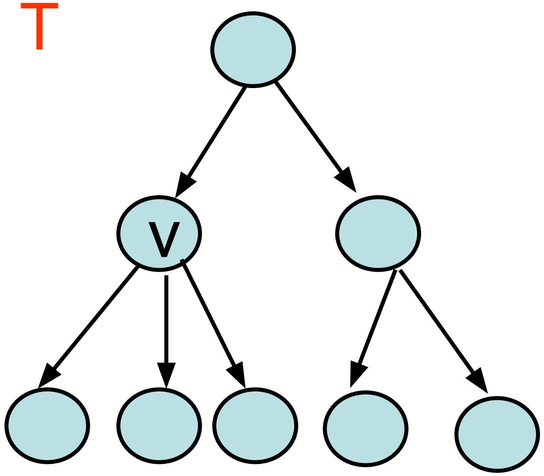
v and w are siblings
u and v are ancestors of x
v and x are descendents of u



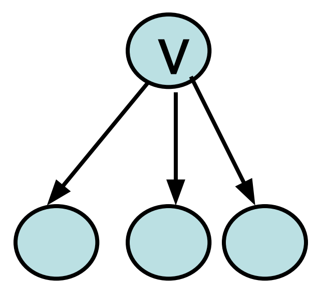
Terminology IV

- A subtree is any node together with all its descendants.

T

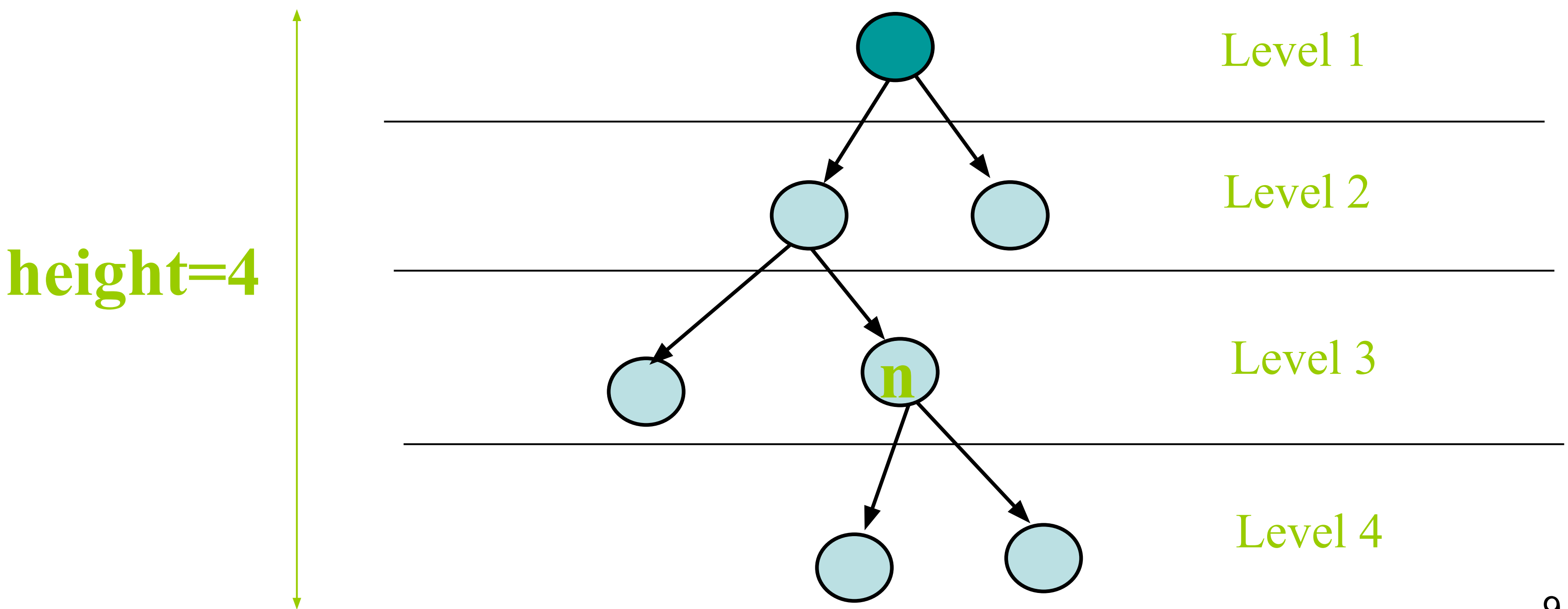


A subtree of T



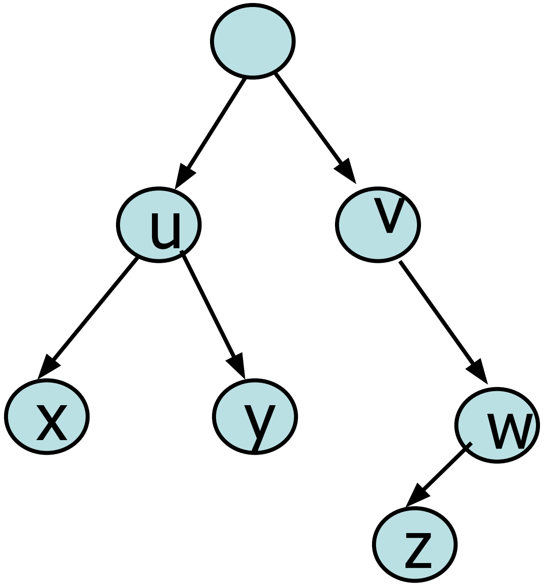
Terminology V

- **Level of a node n:** number of nodes on the path from root to node n
- **Height of a tree:** maximum level among all of its node



Binary Tree

- **Binary Tree:** Every node has at most 2 children
- **Left child of u:** the child on the left of u
- **Right child of u:** the child on the right of u

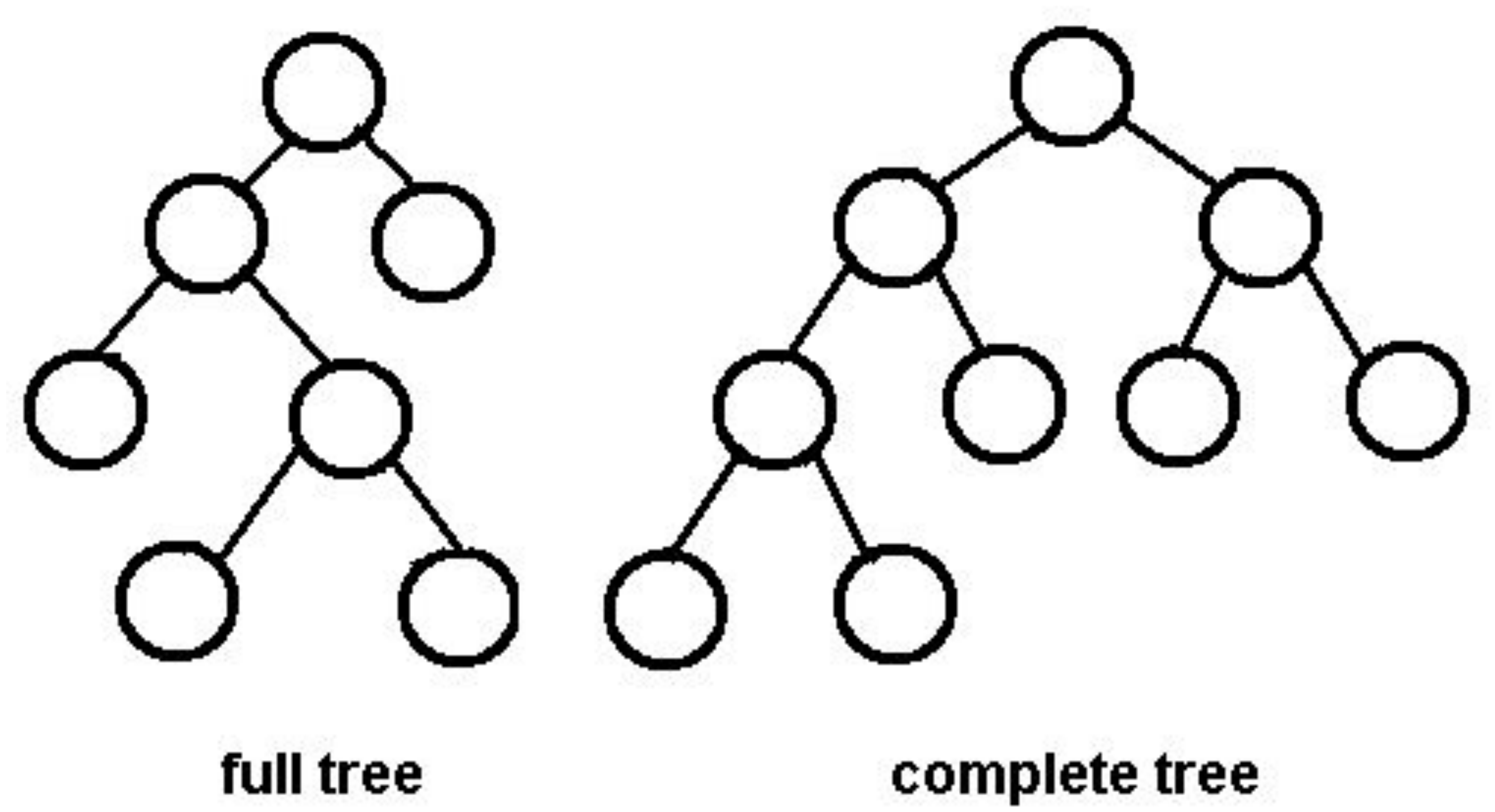


x: left child of u
y: right child of u
w: right child of v
z: left child of w

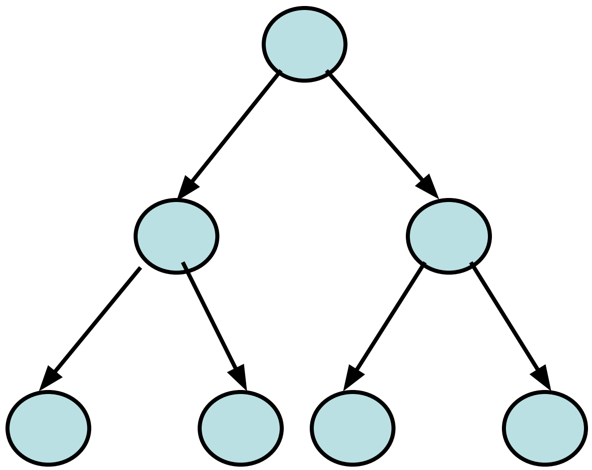
Full and Complete Binary Tree

- T is called a full binary tree, if each node has exactly zero or two children.
- If T is empty, T is a full binary tree of height 0.
- A complete binary tree is a binary tree, which is completely filled, *with the possible exception of the bottom level*, which is filled from **left to right**.

Full vs Complete Binary Tree



Property of binary tree



Level 1: 2^0 nodes

Level 2: 2^1 nodes

Level 3: 2^2 nodes

- A binary tree of height **h** has almost $\mathbf{2^h}$**-1** nodes

No. of nodes $= 2^0 + 2^1 + \ldots + 2^{(h-1)}$

$= 2^h - 1$

Property of binary tree

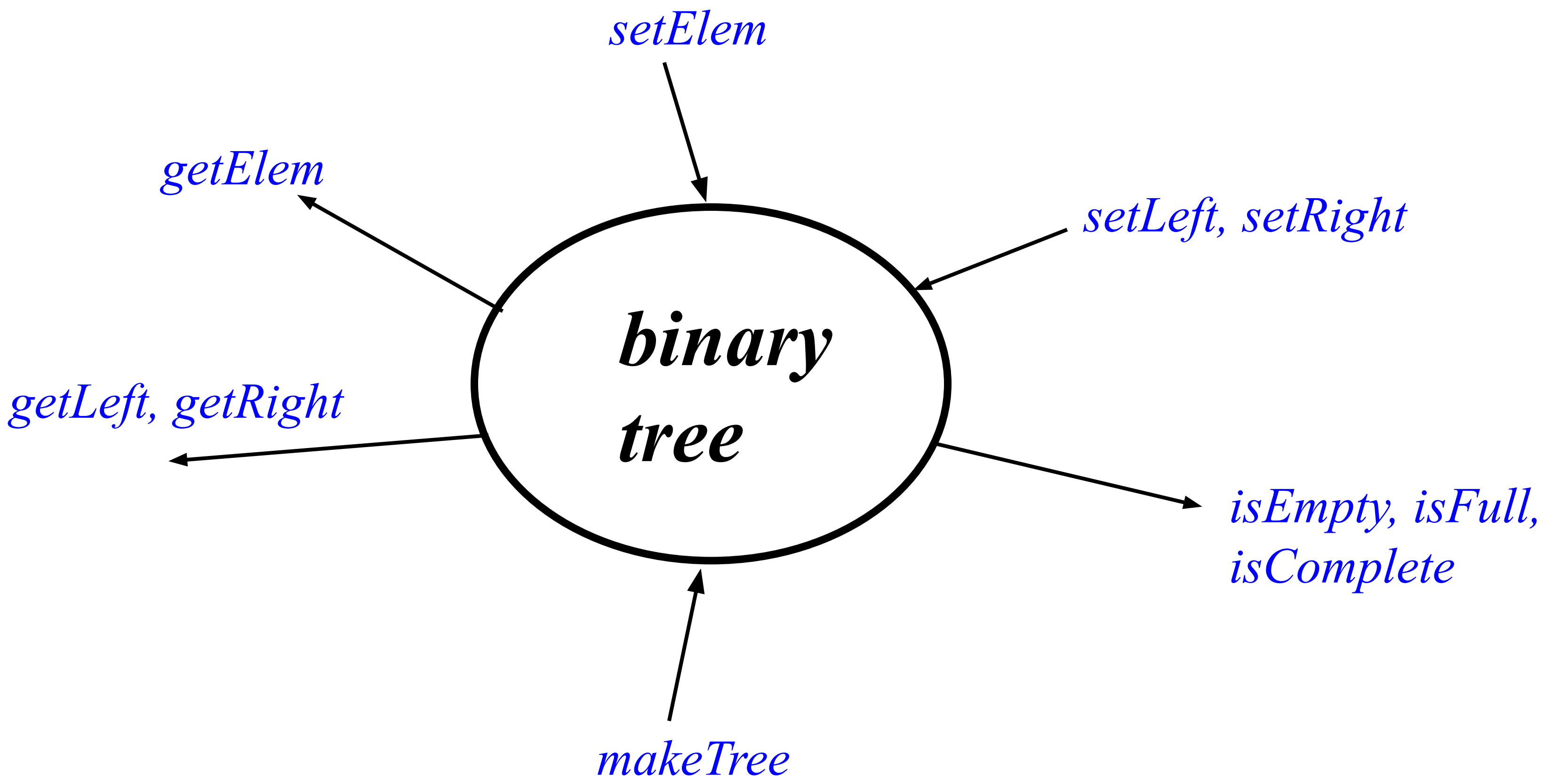
- The minimum height of a binary tree with n nodes is **log(n+1)**

By property, $n \leq 2^h-1$

Thus, $2^h \geq n+1$

That is, $h \geq \log_2(n+1)$

Binary Tree ADT

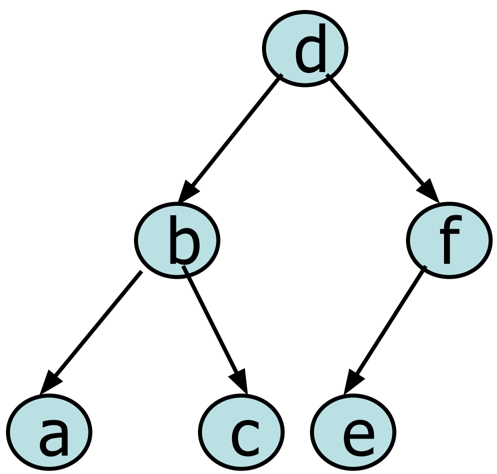


Representation of a Binary Tree

- An array-based representation
- A reference-based representation

An array-based representation

–1: empty tree



nodeNum	item	leftChild	rightChild
0	d	1	2
1	b	3	4
2	f	5	-1
3	a	-1	-1
4	c	-1	-1
5	e	-1	-1
6	?	?	?
7	?	?	?
8	?	?	?
9	?	?	?
...	.....	.....	....

root

0

free

6

Reference Based Representation

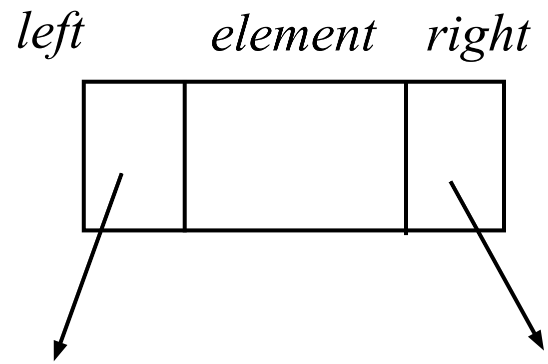
NULL: empty tree

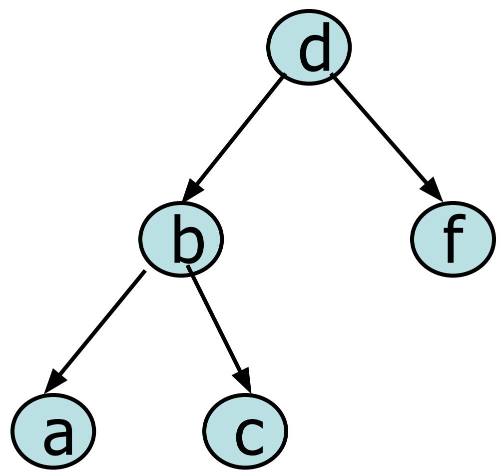
You can code this with a class of three fields:

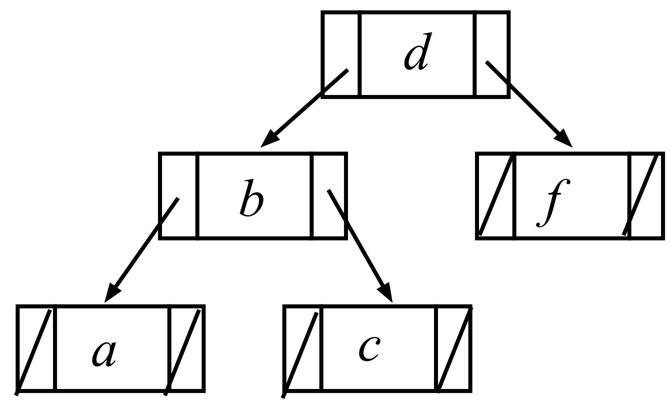
Object element;

BinaryNode left;

BinaryNode right;







Tree Traversal

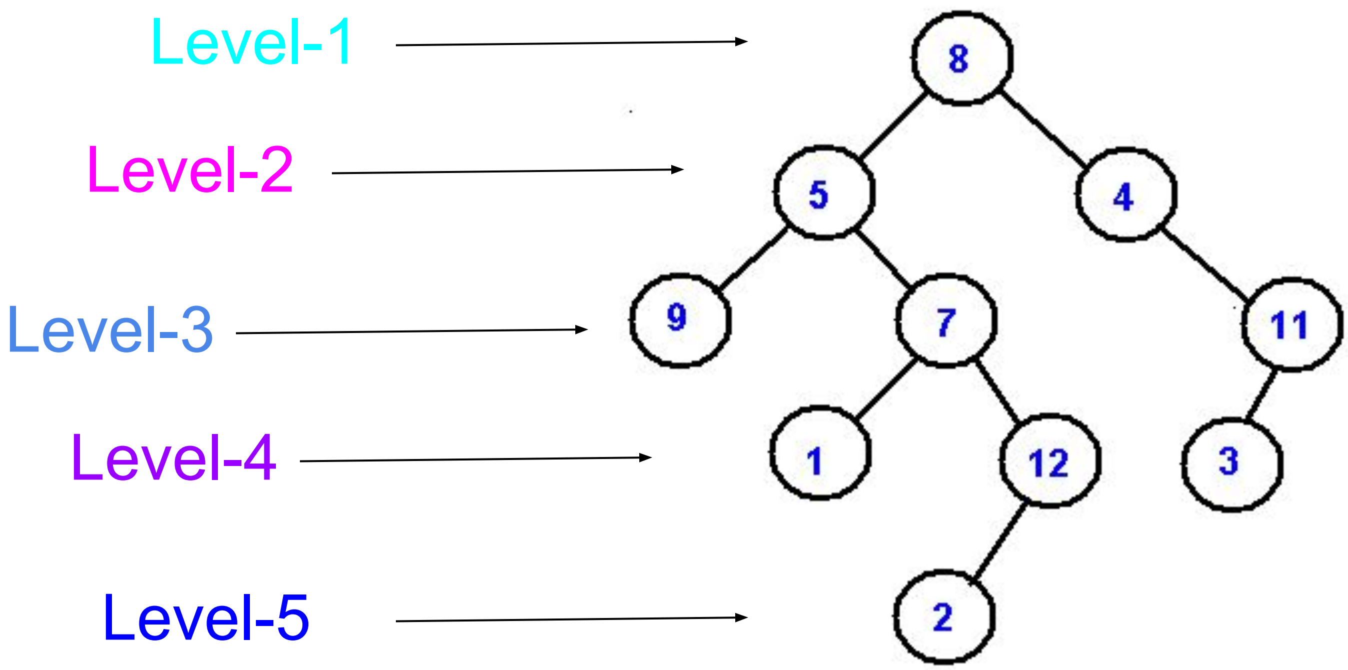
- Given a binary tree, we may like to do some operations on all nodes in a binary tree.
- **For example, we may want to double the value in every node in a binary tree.**
- To do this, we need a traversal algorithm which visits every node in the binary tree.

Ways to traverse a tree

- **Pre-order:**
  - (1) visit node
  - (2) recursively visit left subtree
  - (3) recursively visit right subtree
- **In-order:**
  - (1) recursively visit left subtree
  - (2) visit node
  - (3) recursively visit right subtree
- **Post-order:**
  - (1) recursively visit left subtree
  - (2) recursively visit right subtree
  - (3) visit node

Level-Order

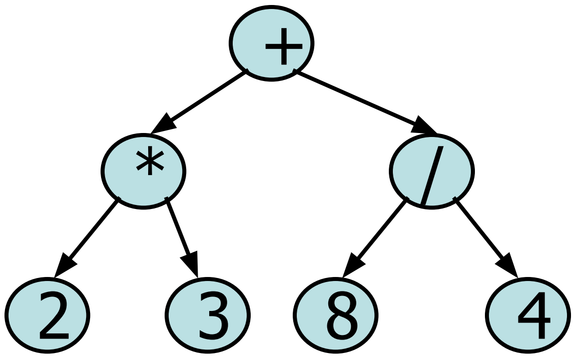
Traverse the nodes level by level (Left to Right)



LevelOrder: 8, 5, 4, 9, 7, 11, 1, 12, 3, 2

Examples for expression tree

- By pre-order, (prefix)
  + * 2 3 / 8 4
- **By in-order, (infix)**
  2 * 3 + 8 / 4
- By post-order, (postfix)
  2 3 * 8 4 / +
- By level-order,
  + * / 2 3 8 4
- **Note 1:** Infix is what we read!
- **Note 2:** Postfix expression can be computed efficiently using stack



Pre-order

Algorithm pre-order(BTree x)

```
If (x is not empty) {
    print x.getItem();          // you can do other things!
    pre-order(x.getLeftChild());
    pre-order(x.getRightChild());
}
```

Pre-order

Pre-order(a);

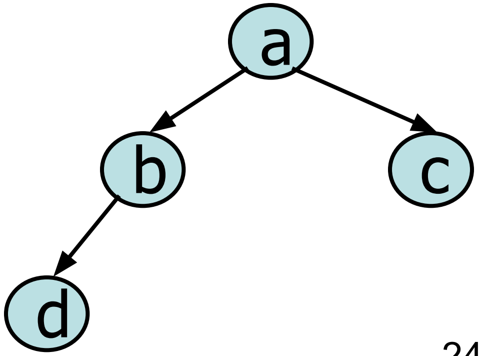
Print a;
Pre-order(b);
Pre-order(c);

Print b;
Pre-order(d);
Pre-order(null);

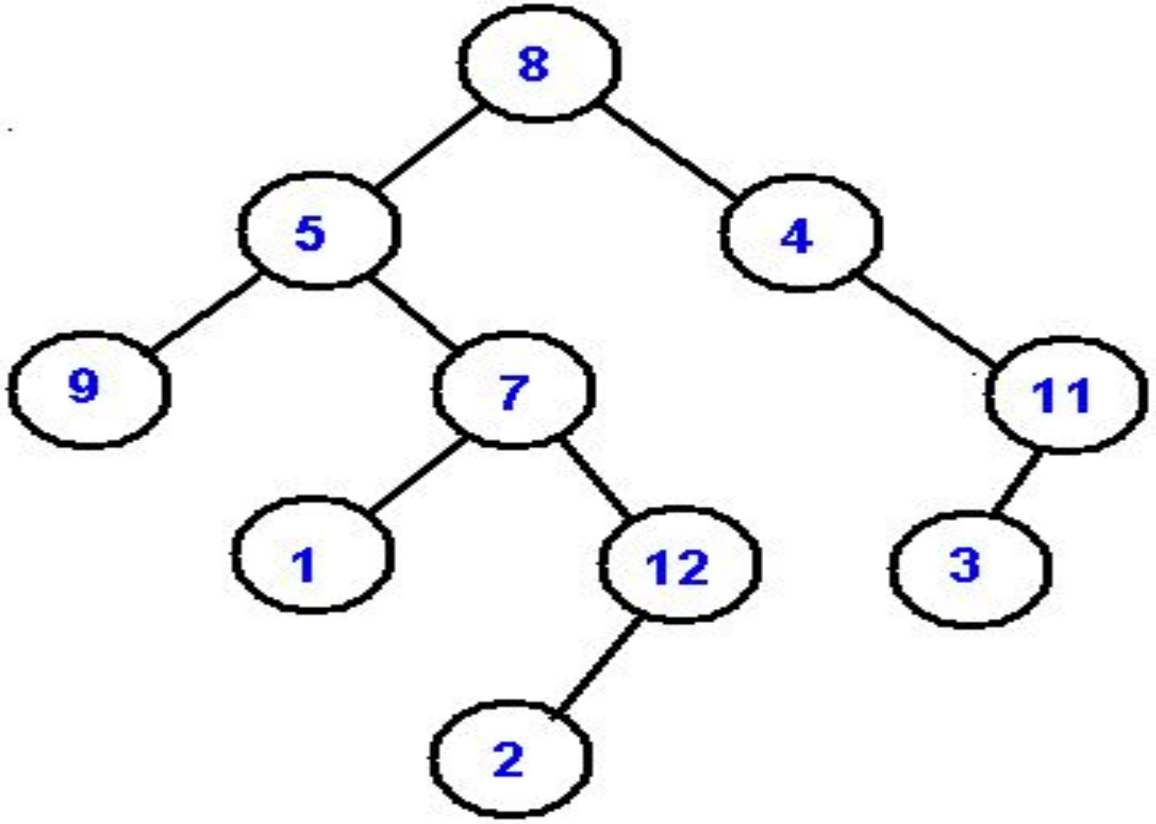
Print d;
Pre-order(null);
Pre-order(null);

Print c;
Pre-order(null);
Pre-order(null);

a b d c



Pre-order Example



Pre-Order: 8, 5, 9, 7, 1, 12, 2, 4, 11, 3

Time complexity of Pre-order Traversal

- For every node x, we will call *pre-order(x)* one time, which performs O(1) operations.
- Thus, the total time = O(n).

In-order and post-order

```
Algorithm in-order(BTree x)
If (x is not empty) {
    in-order(x.getLeftChild());
    print x.getItem(); // you can do other things!
    in-order(x.getRightChild());
}

Algorithm post-order(BTree x)
If (x is not empty) {
    post-order(x.getLeftChild());
    post-order(x.getRightChild());
    print x.getItem(); // you can do other things!
}
```

In-order

Algorithm in-order(BTree x)

```
If (x is not empty) {
    in-order(x.getLeftChild());
    print x.getItem(); // you can do other things!
    in-order(x.getRightChild());
}
```

In-order(a);

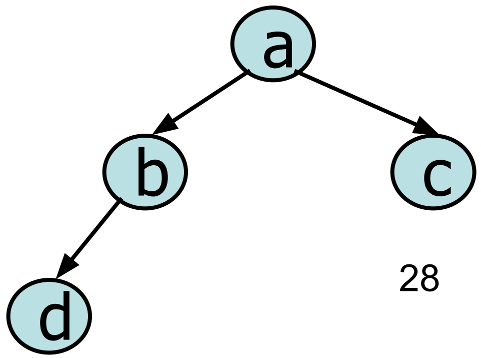
In-order(b);
Print a;
In-order(c);

In-order(d);
Print b;
In-order(null);

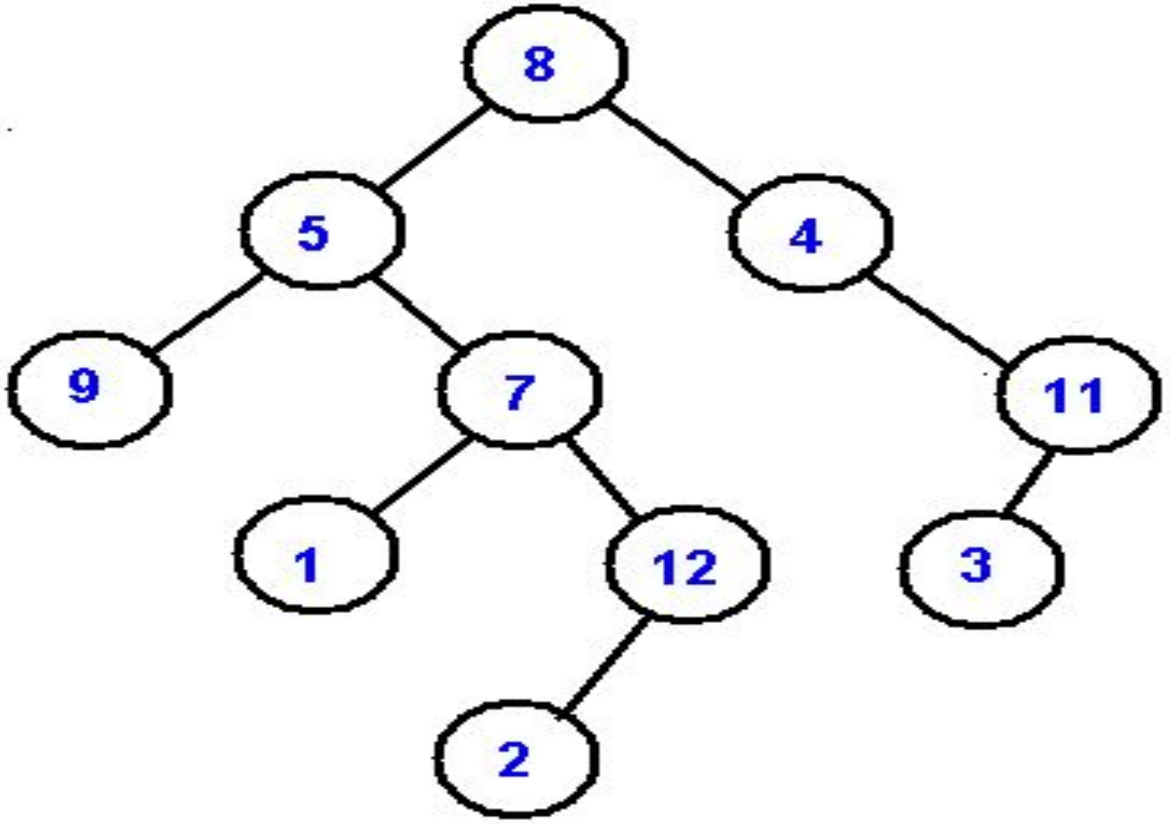
In-order(null);
Print d;
In-order(null);

In-order(null);
Print c;
In-order(null);

d b a c



In-order Example



In-Order: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11

Post-order

Post-order(a);

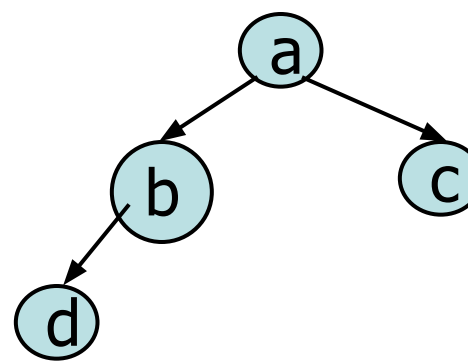
Post-order(b);
Post-order(c);
Print a;

Post-order(d);
Post-order(null);
Print b;

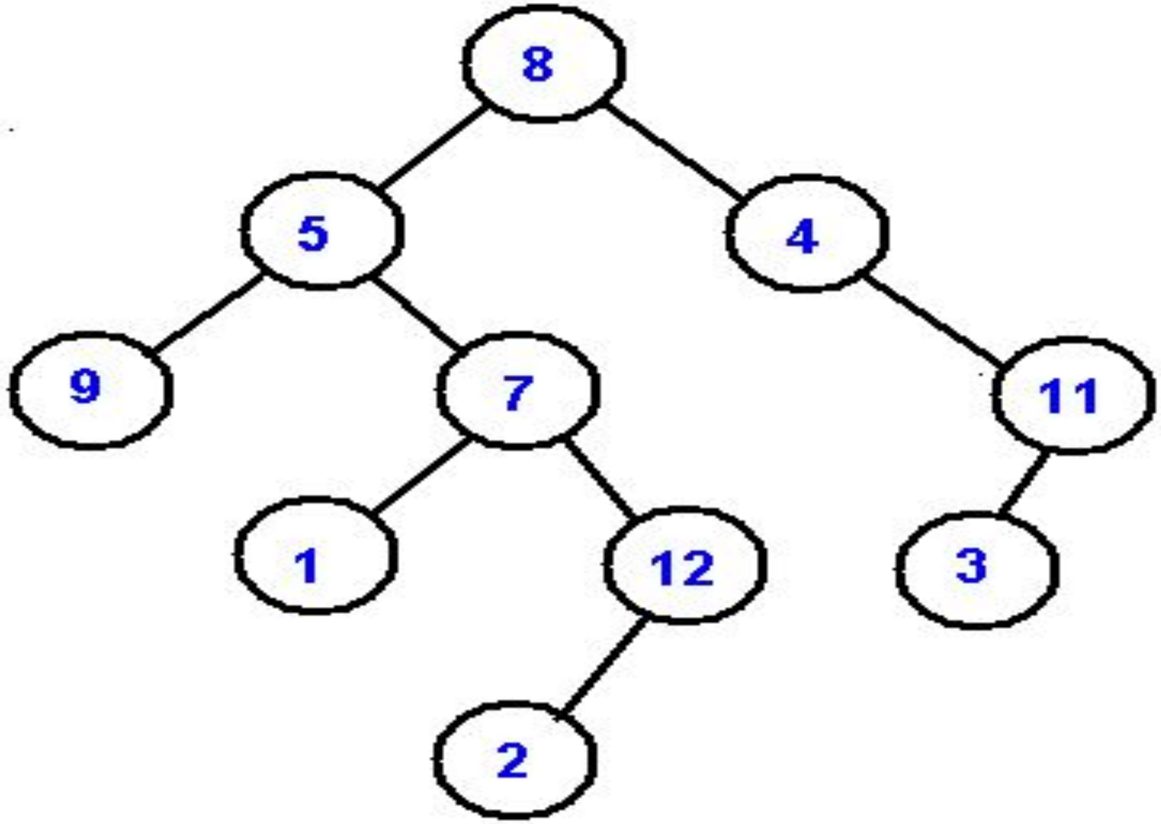
Post-order(null);
Post-order(null);
Print d;

Post-order(null);
Post-order(null);
Print c;

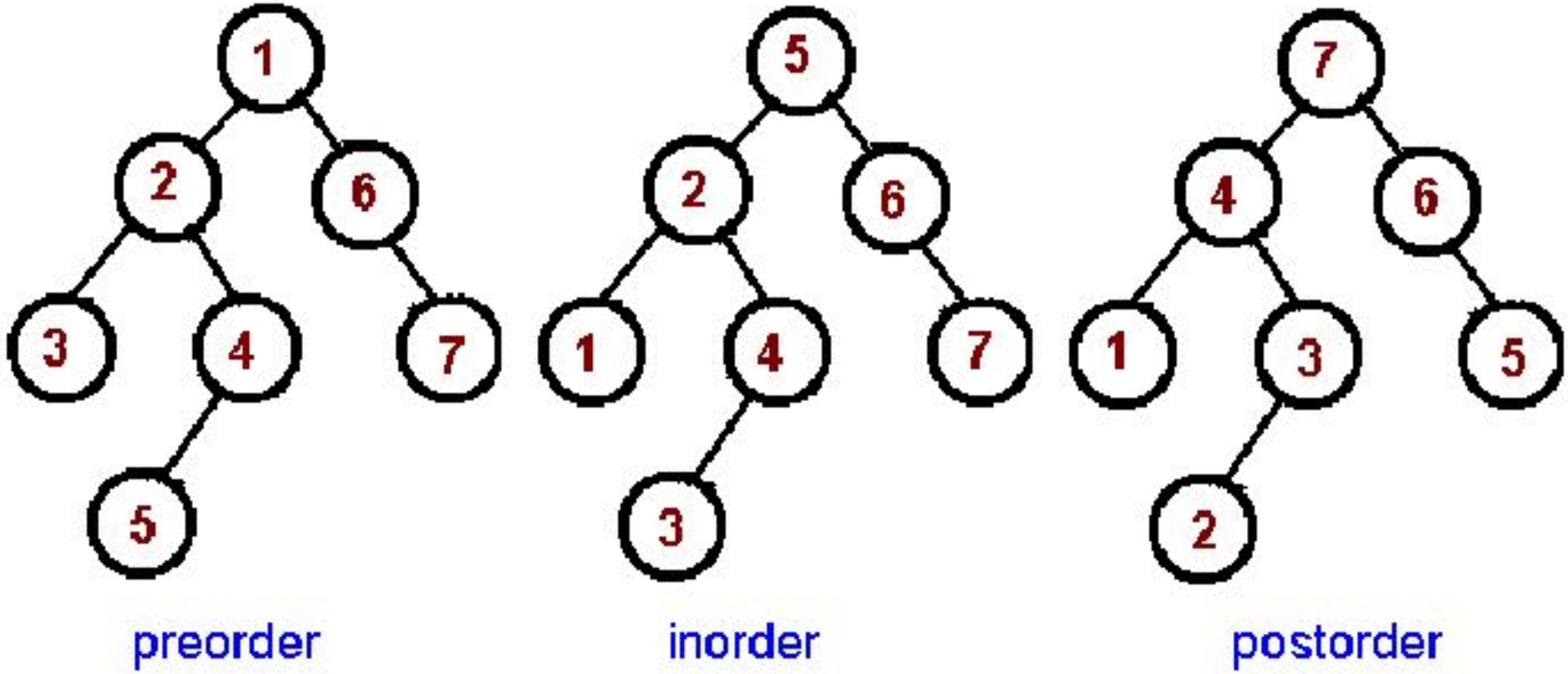
d b c a

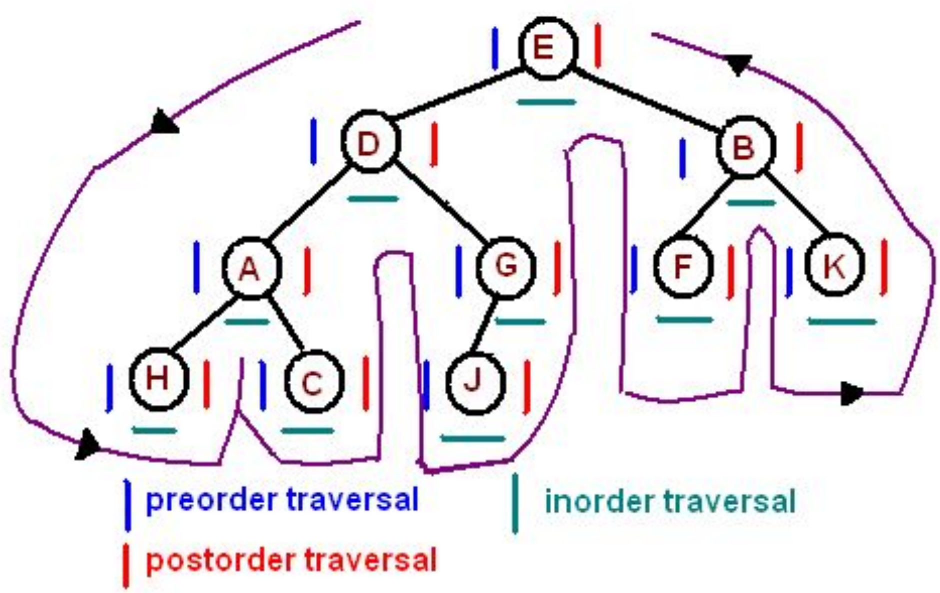


Post-order Example



Post-Order: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8


1
2
6
3
4
7
5
preorder
5
2
6
1
4
7
3
inorder
7
4
6
1
3
5
2
postorder

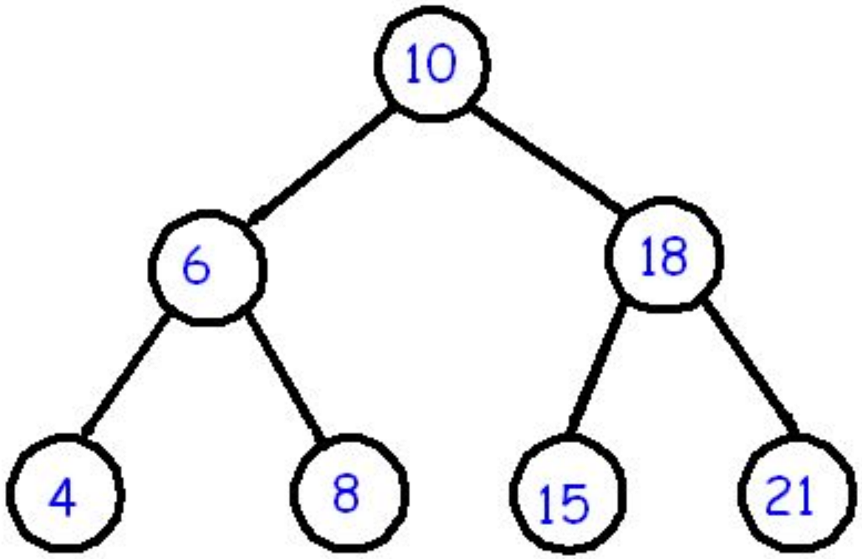

E
D
B
A
G
F
K
H
C
J
preorder traversal
inorder traversal
postorder traversal

Time complexity for In-order and Post-order

- Similar to pre-order traversal, the time complexity is $O(n)$.

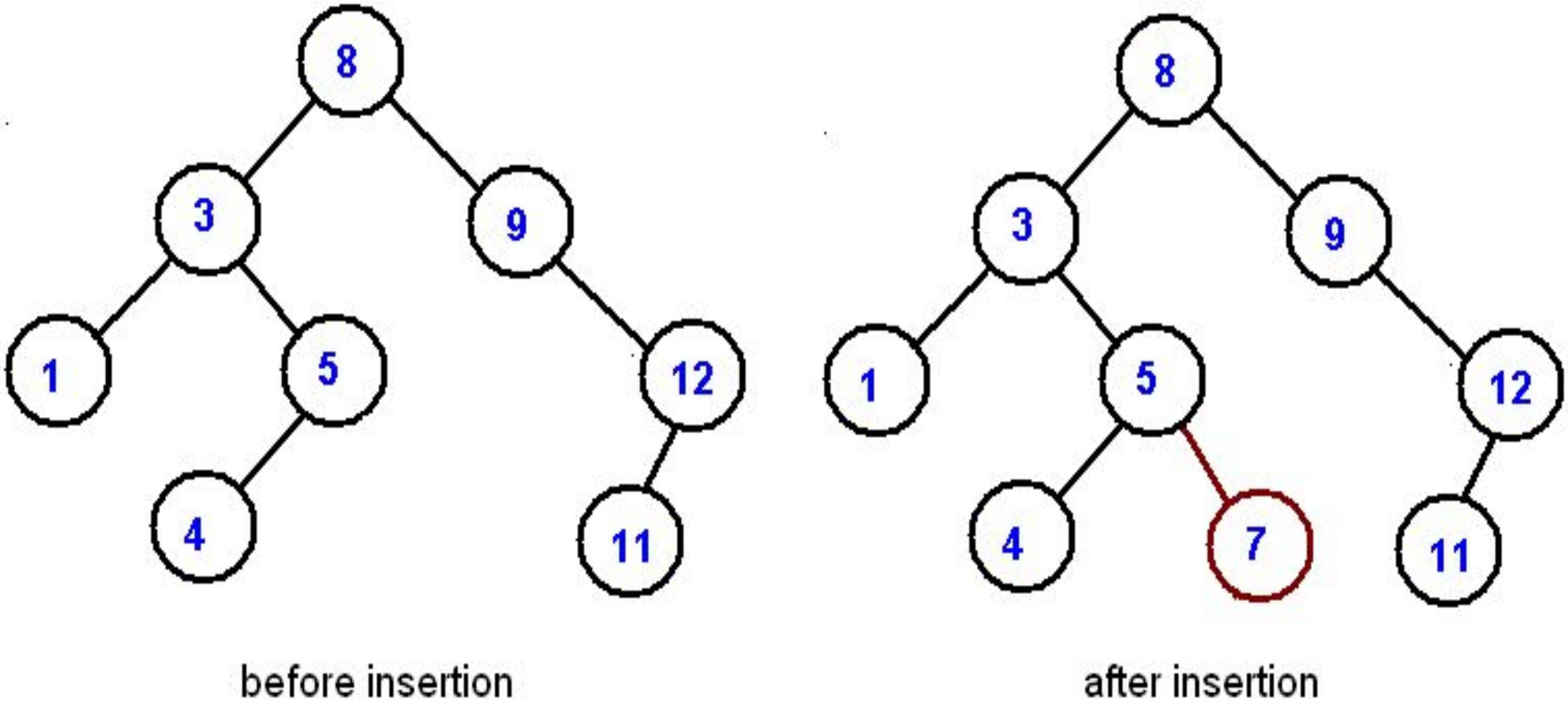
Binary Search Trees

- Each node contains one key (known as **data**)
- The keys in the left subtree are less then the key in its parent node, in short L < P;
- The keys in the right subtree are greater the key in its parent node, in short P < R;
- **Duplicate keys are not allowed**



Insertion

(Insert 7)



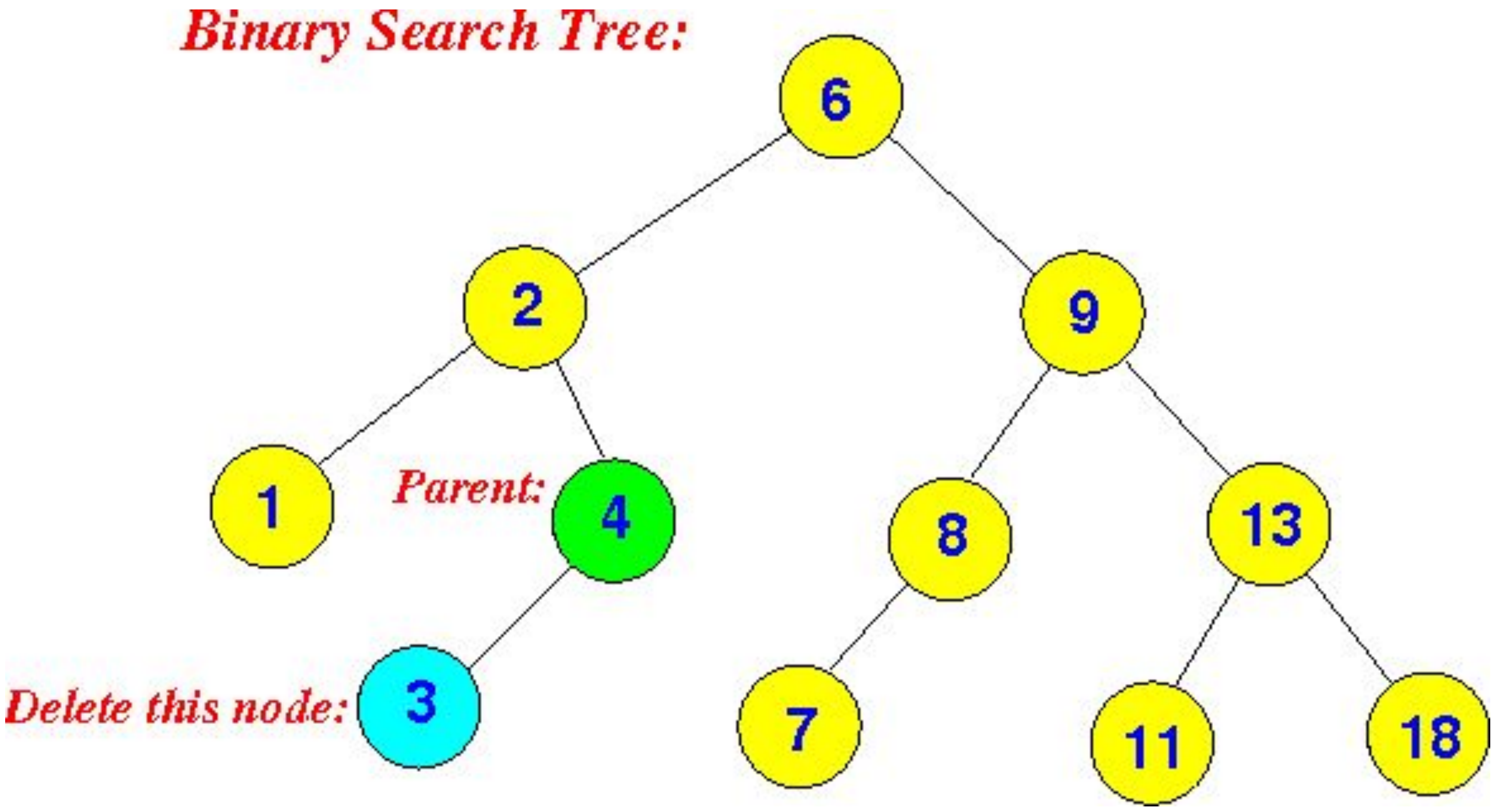
1. Searching element in LL
2. Complexity O(n)

1. Searching element in BST
2. Complexity O(log n)

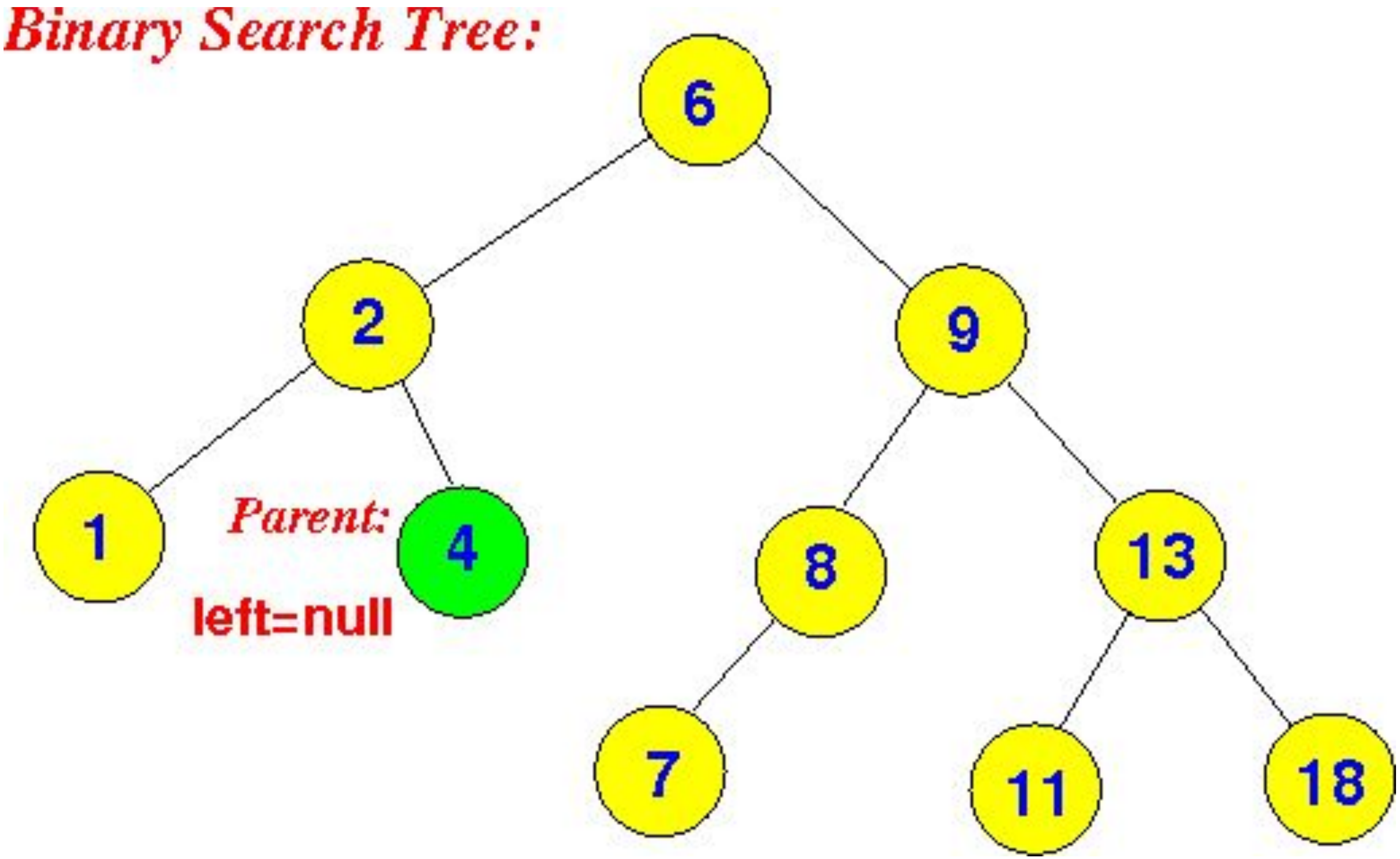
Deletion

- is not in a tree ?
- is a leaf ?
- has only one child ?
- has two children ?

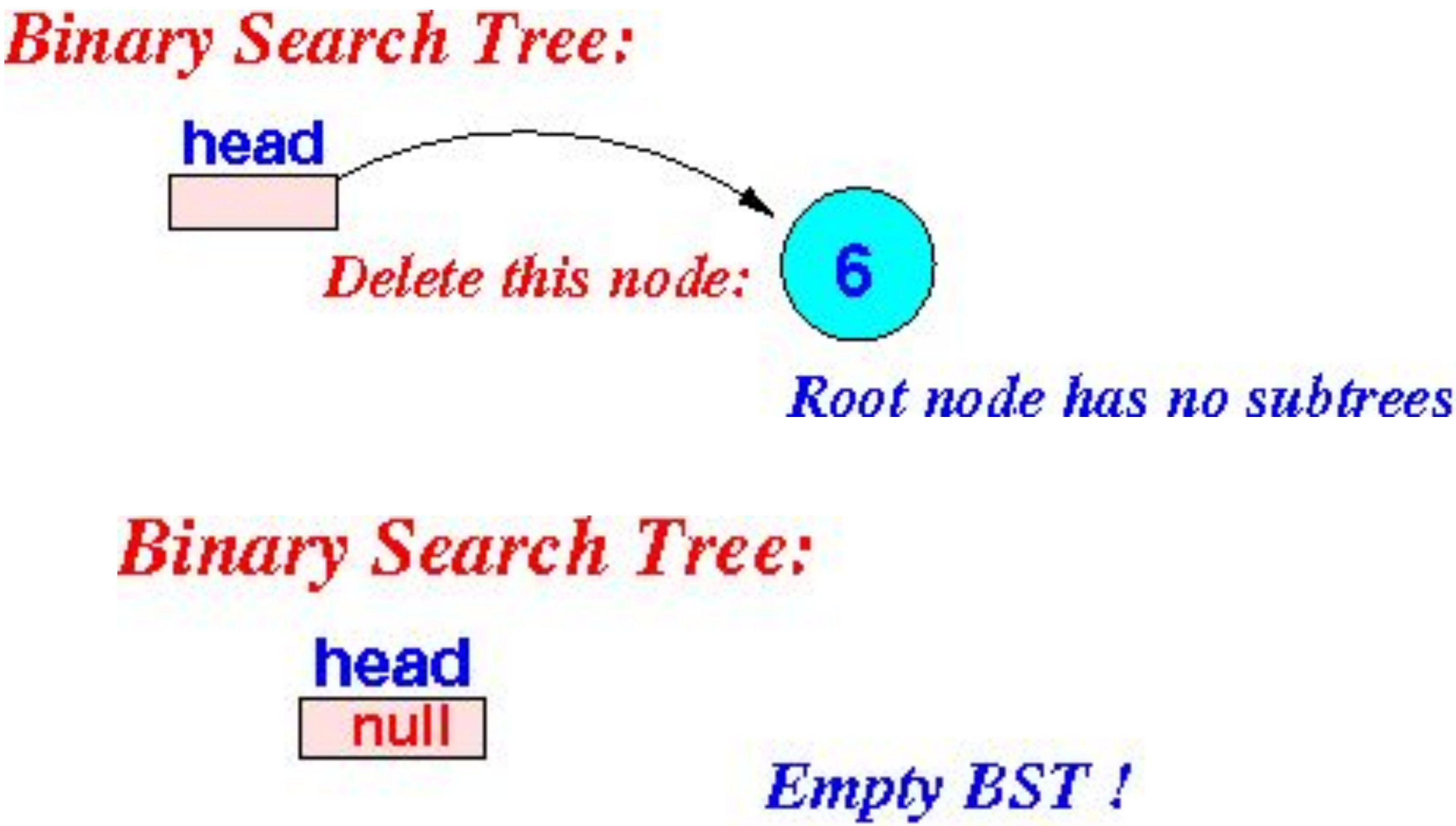
Delete the **node 3** in the following **BST**:



Resulting BST (After delete 3)

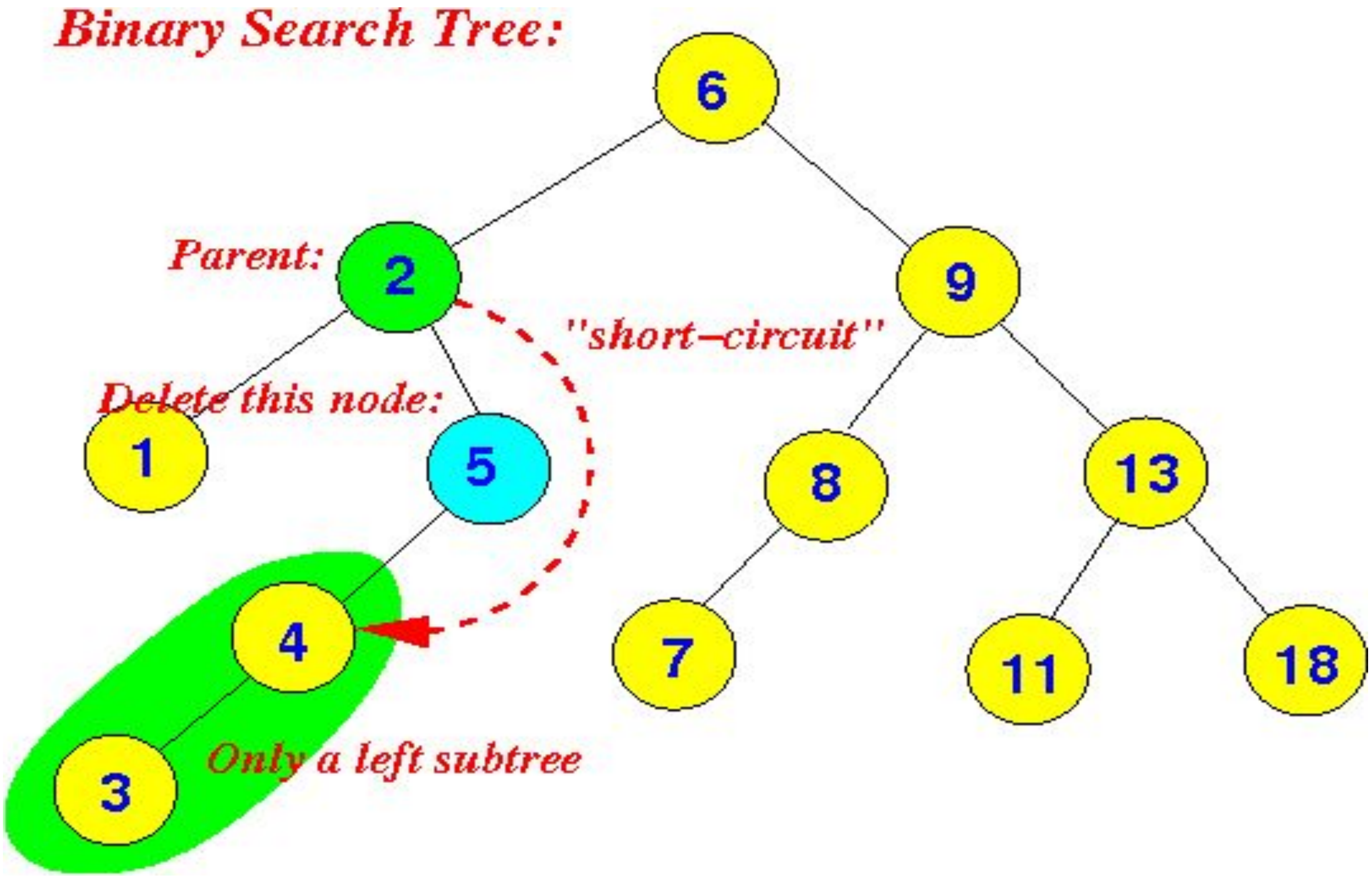


Delete root node



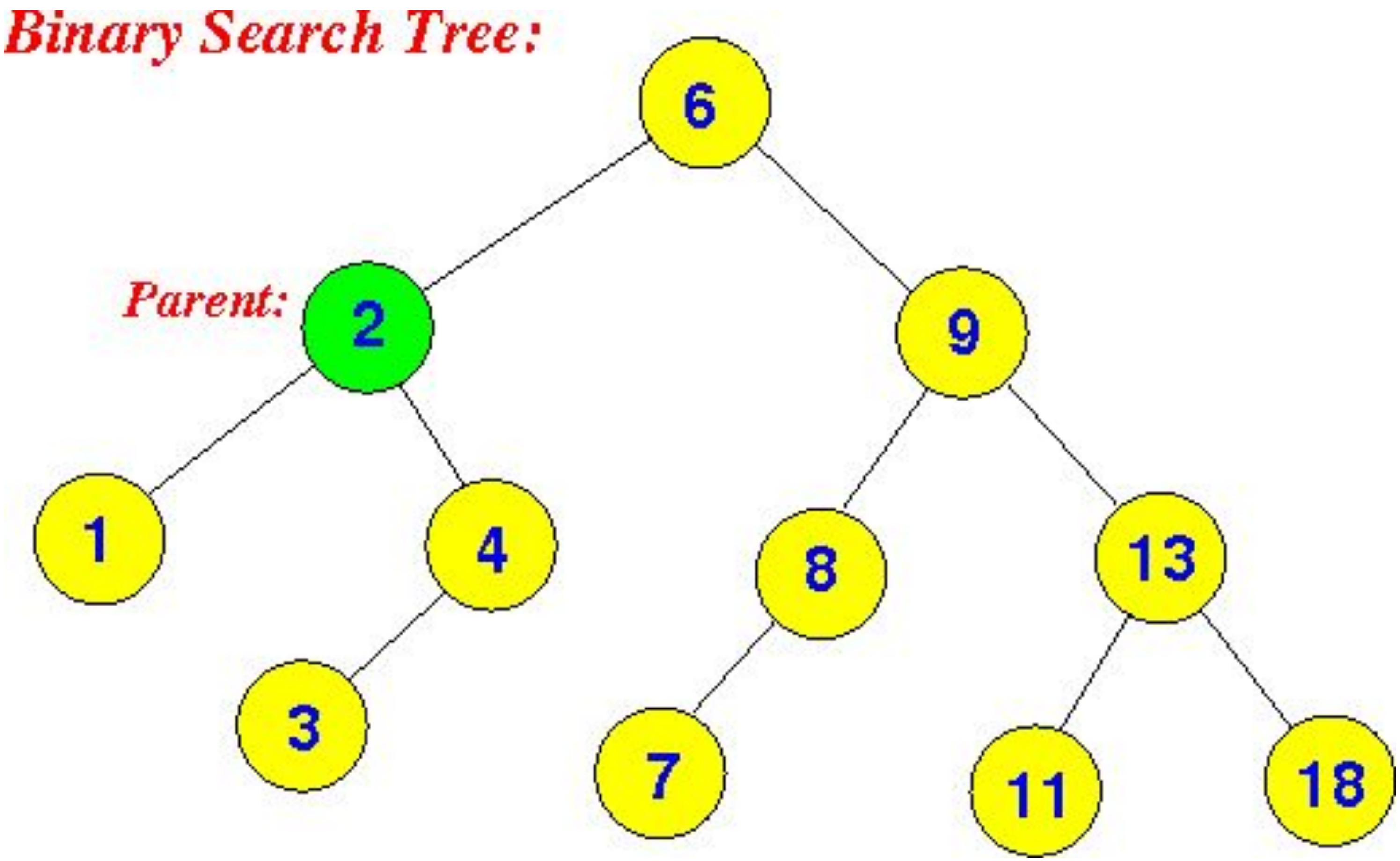
Deletion node *only* has a *left* subtree

Delete the **node 5** in the following **BST**:



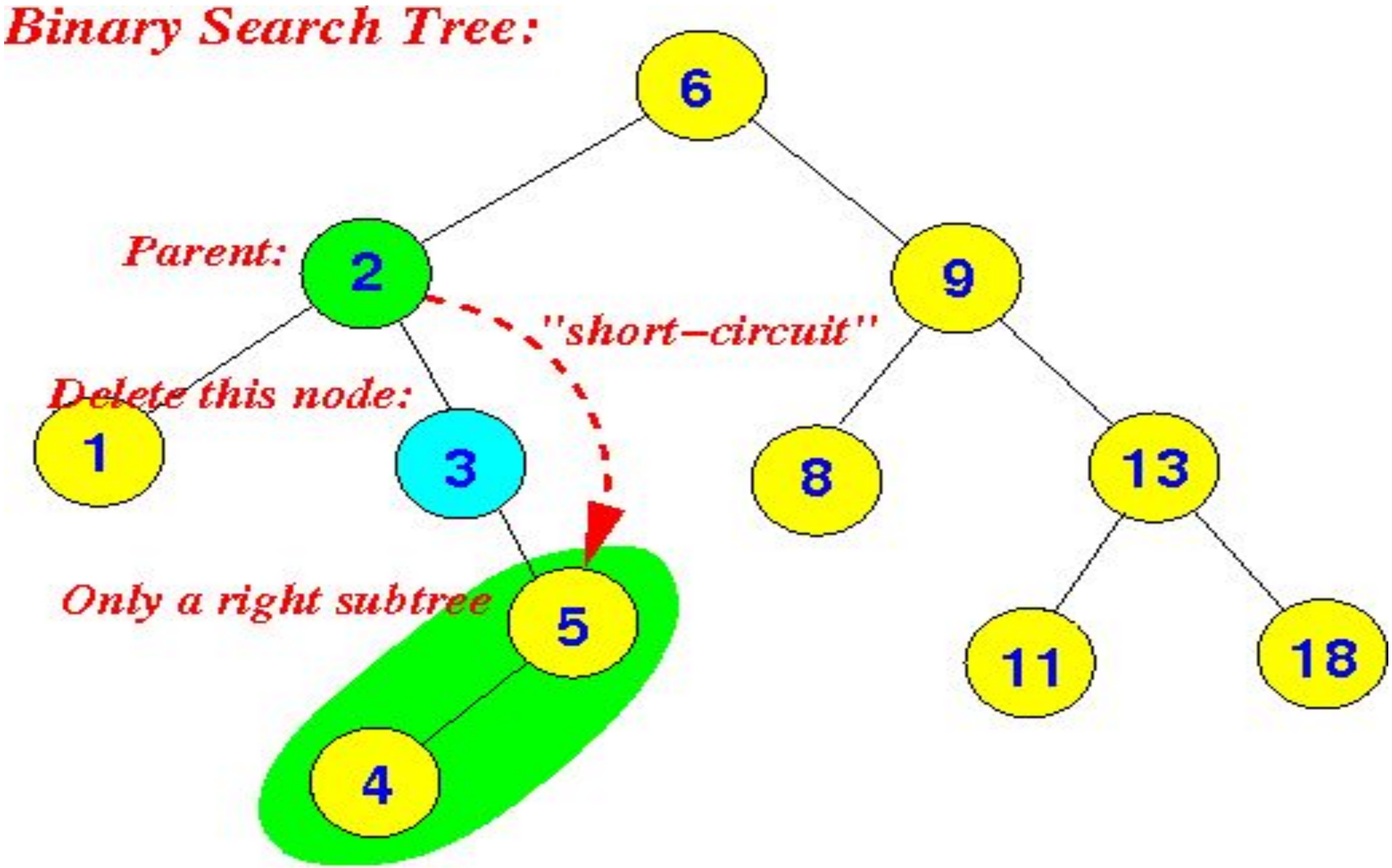
Deletion node *only* has a *left* subtree

Resulting after Delete the node 5

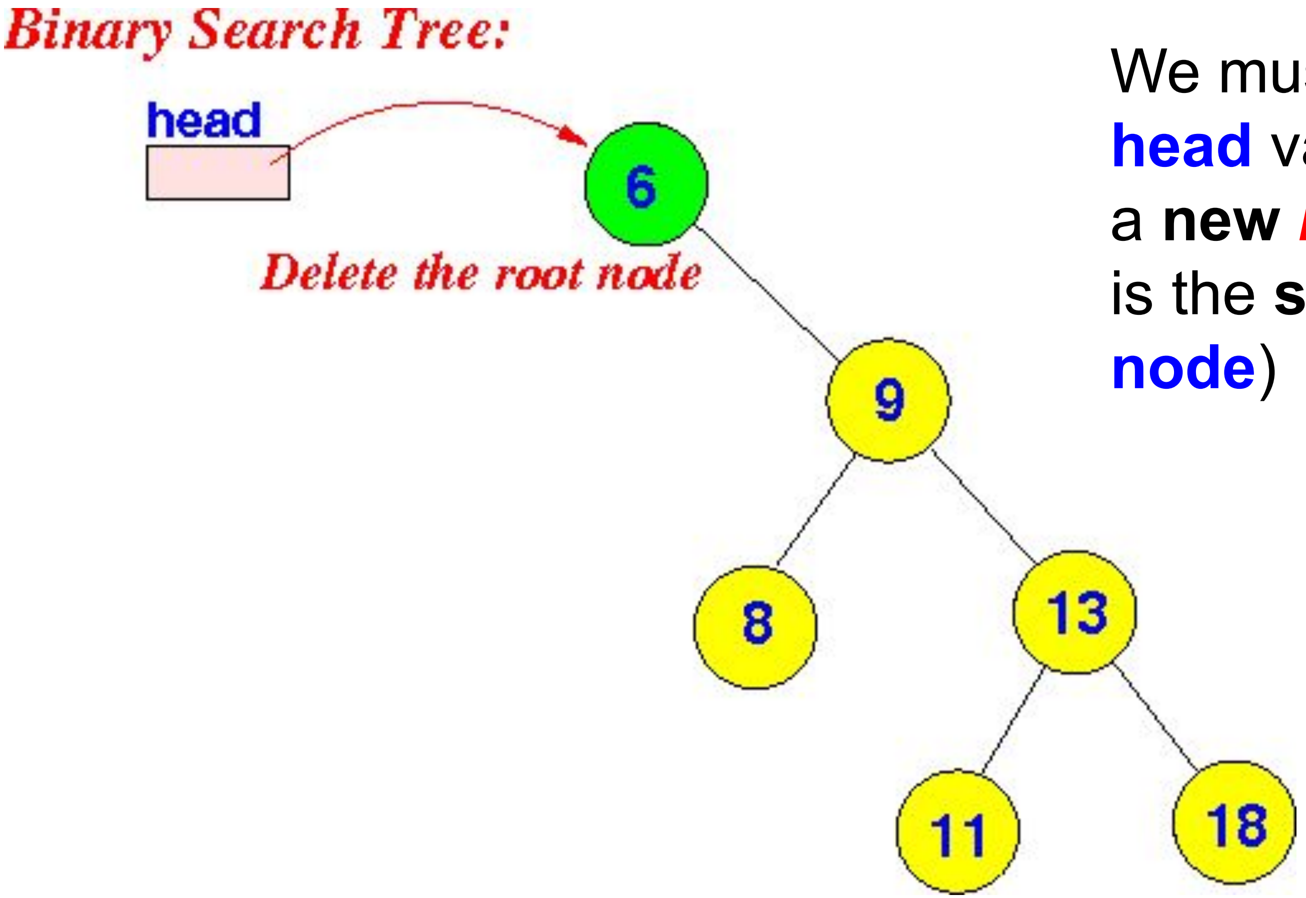


Deletion node *only* has a *right* subtree

Delete the **node 18** in the following **BST**:

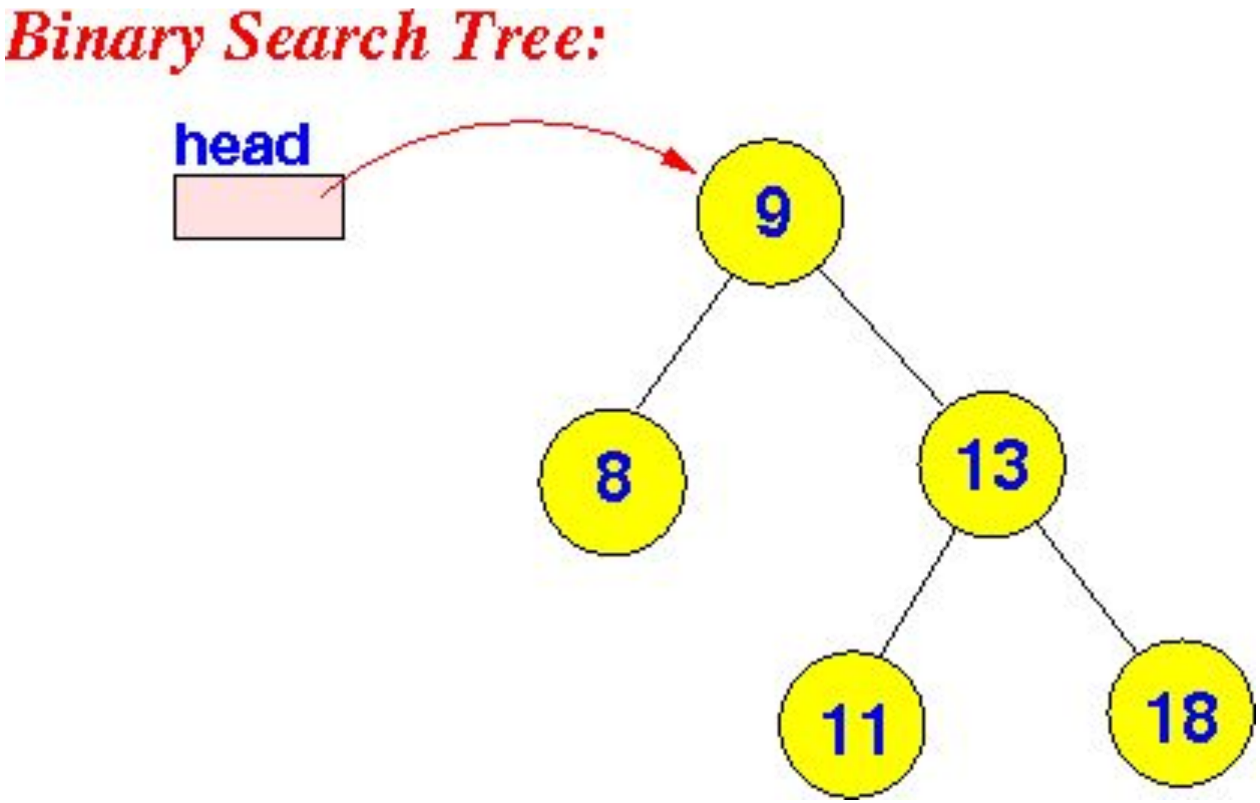


Special situation: the deletion node is the *root* node



We must **update** the **head** variable to point to a **new** ***root* node** (which is the **subtree** of the **root node**)

Special situation: the deletion node is the *root* node

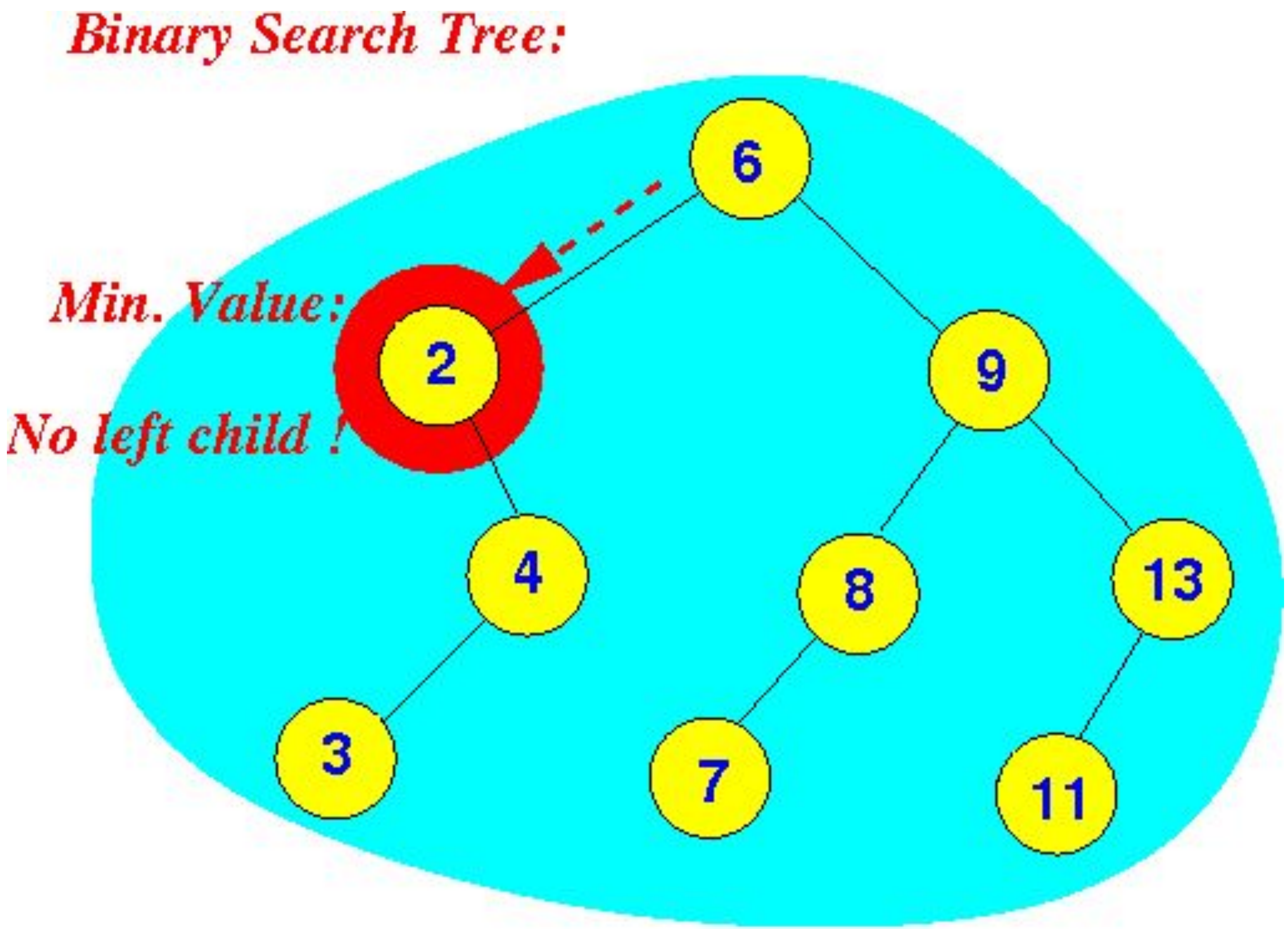


Deleting a node that has *two* subtrees

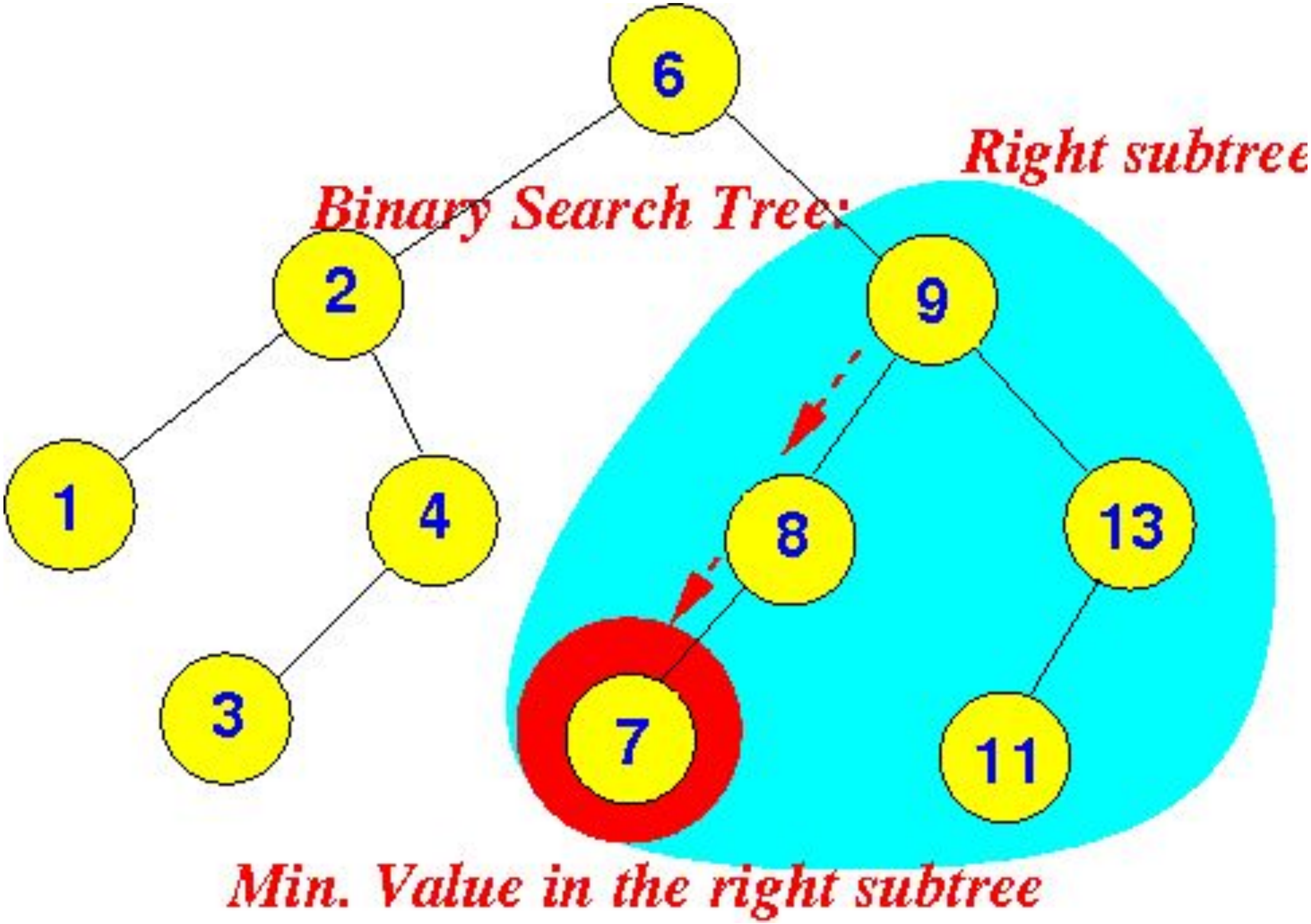
- **First**, we **find** the **deletion node *p*** (= the **node** that we want to **delete**)
- ***Find*** the ***successor* node** of ***p***
- **Replace** the **content** of node ***p*** with the **content** of the **successor node**
- **Delete** the **successor node**

Finding the *successor* node

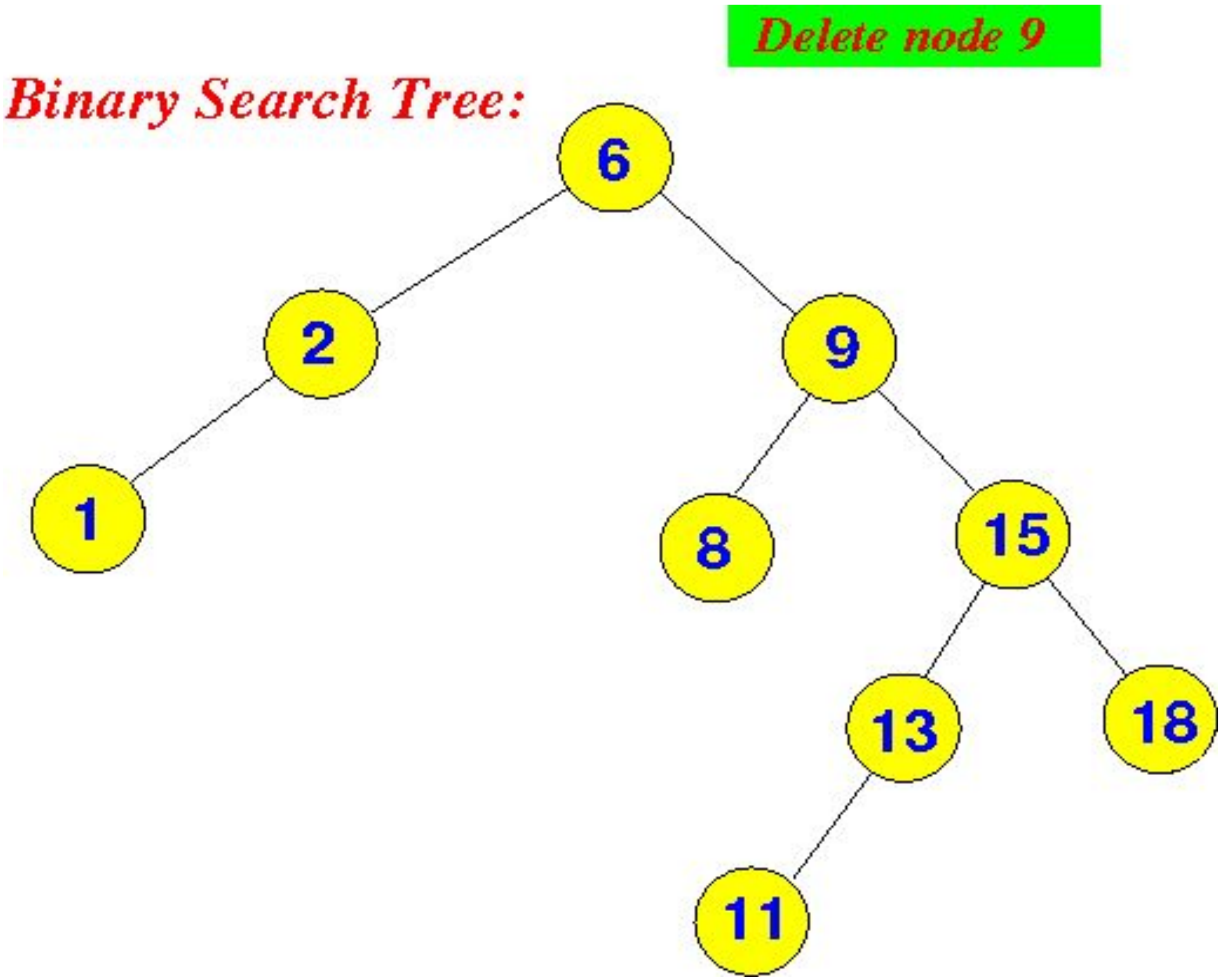
- **Successor node** is the **node** in the ***right* subtree** that has the ***minimum* value**



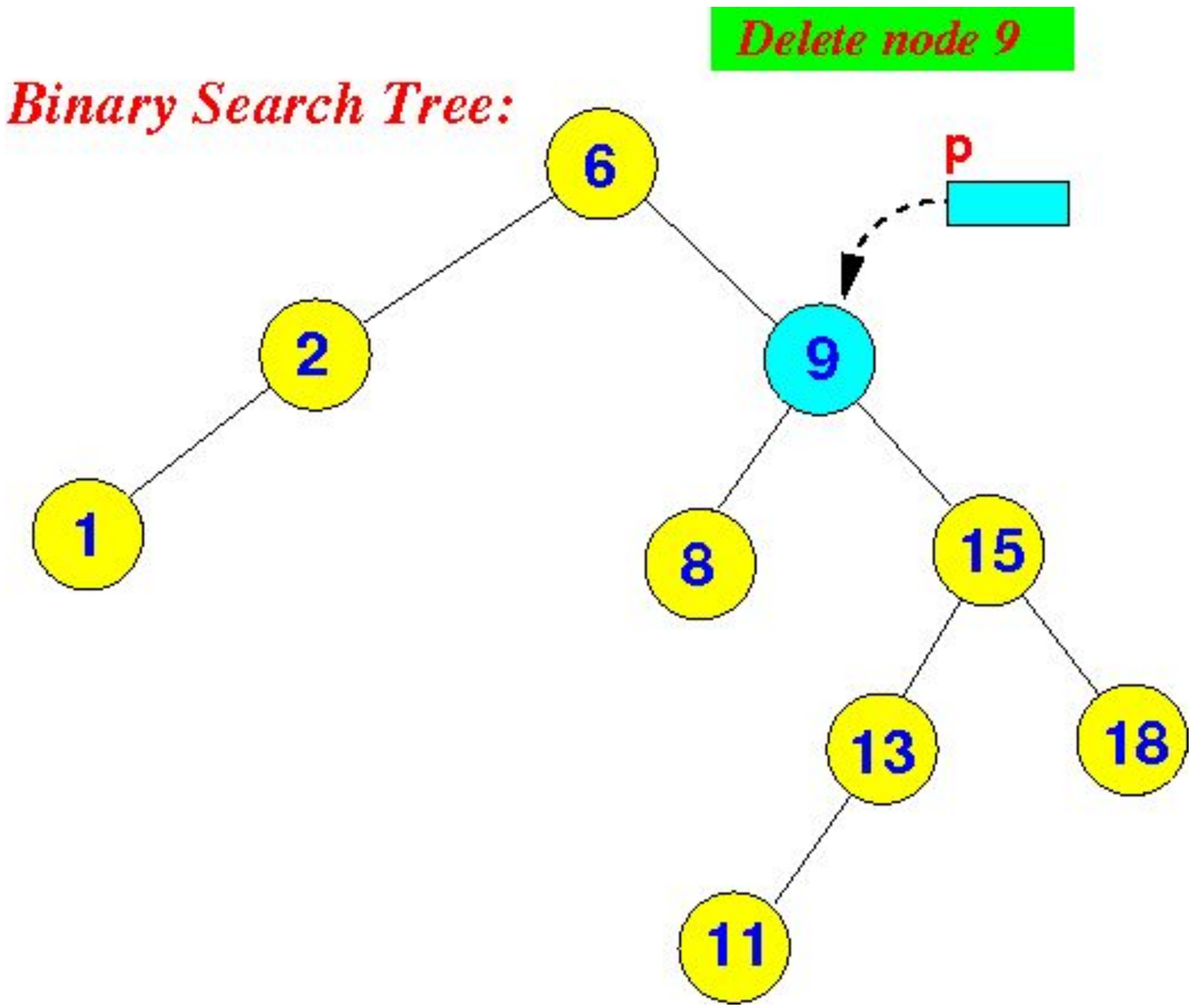
Min. Value in Right Subtree



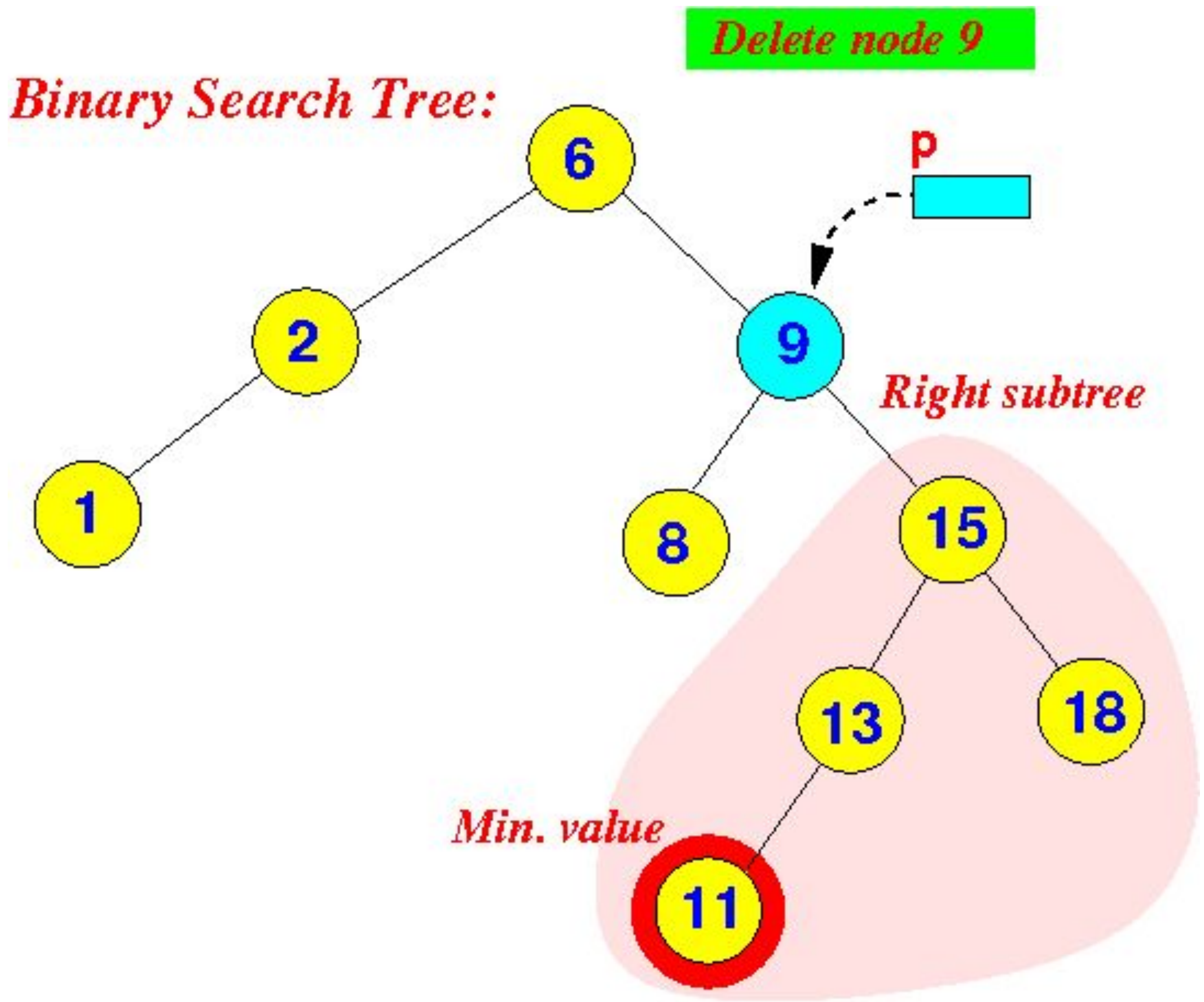
Deleting a node with 2 subtrees



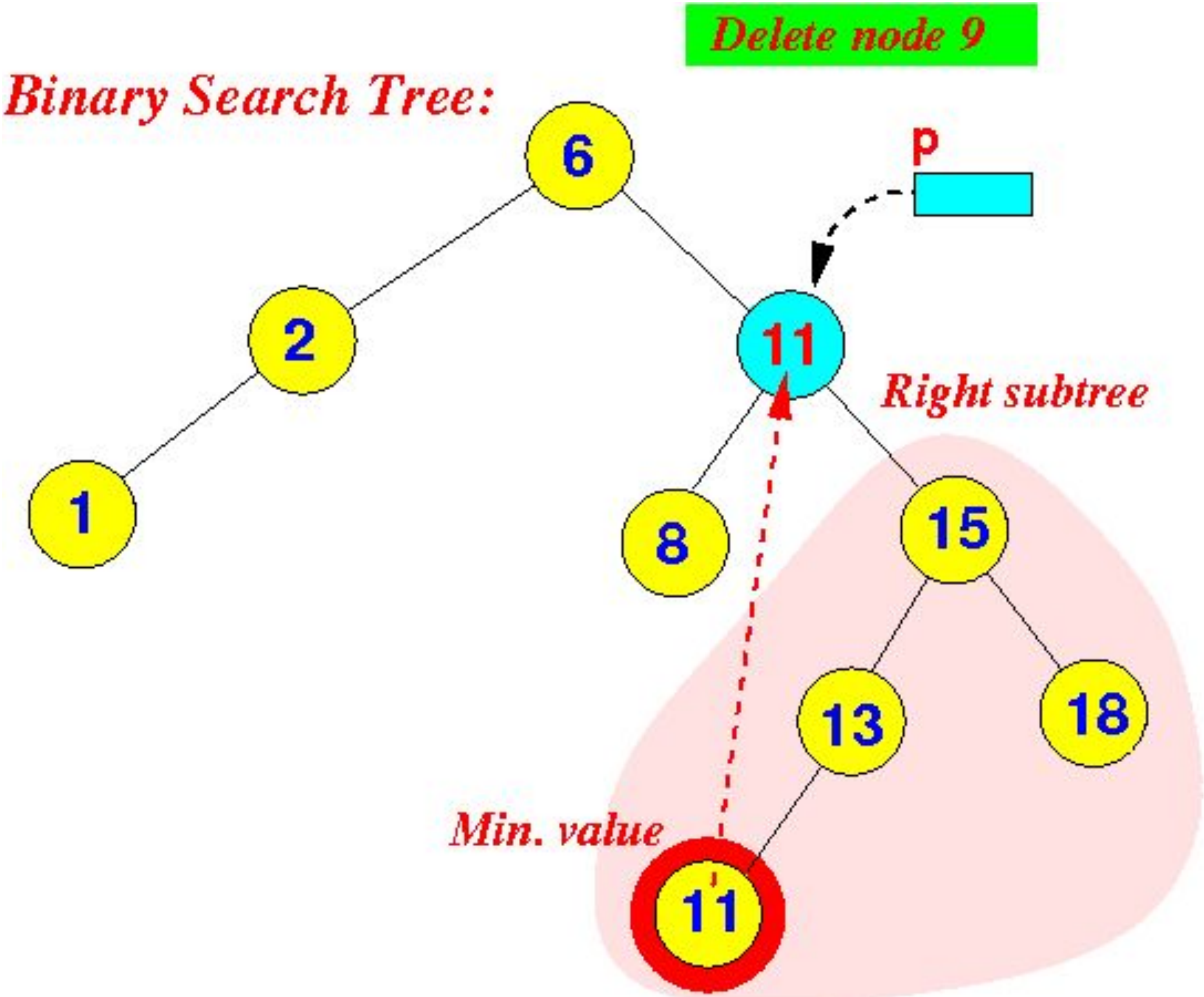
1. we must find the node with the value 9:



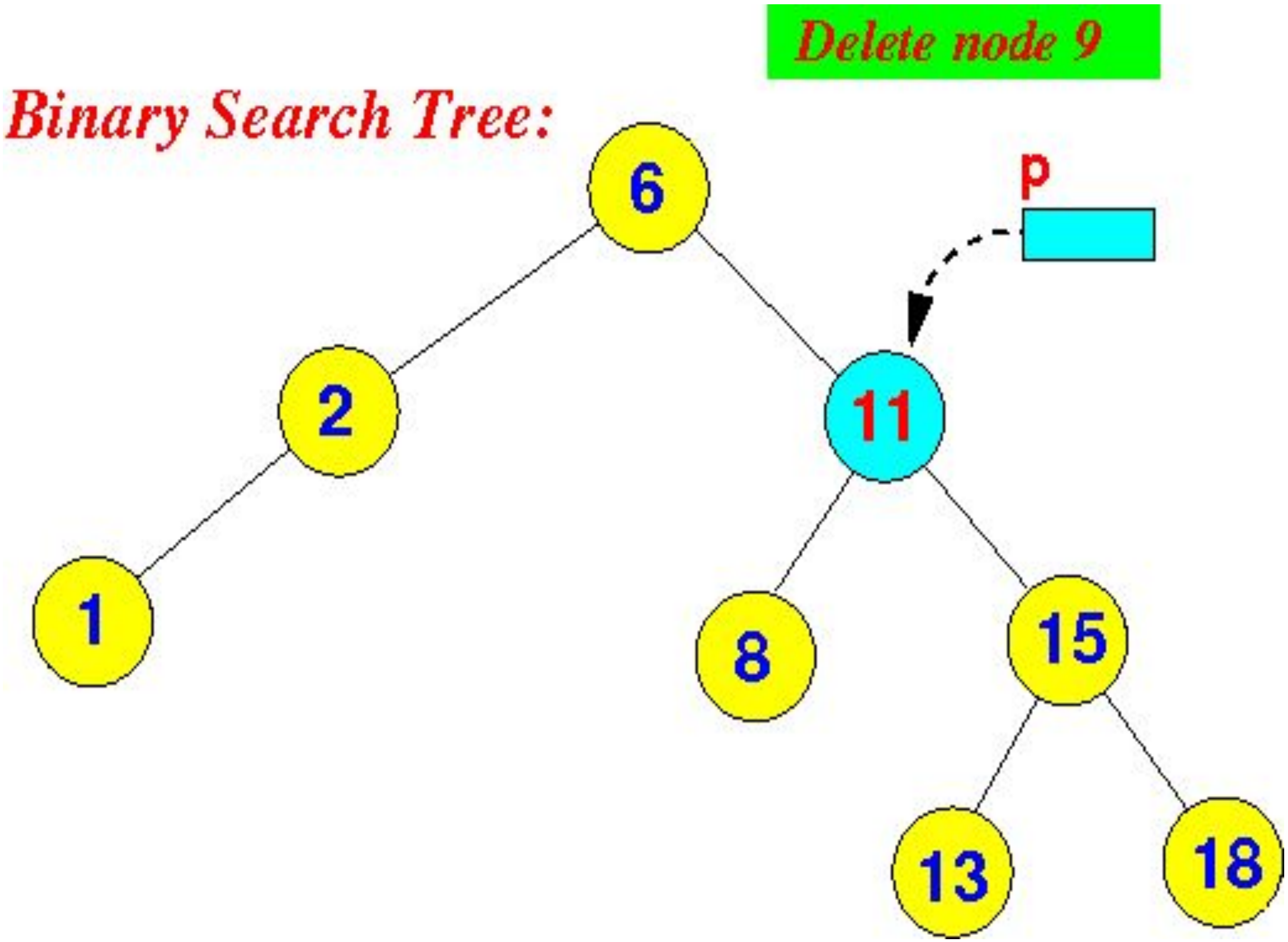
2. we find the *successor* node of the node



3. We copy the content of the *successor* node of the node into the *deletion* node (p):



4. **Last step** is **deleting** the ***successor* node**



Notice that the tree satisfies the **Binary Search Tree** property

Delete a node with *2* subtrees --- pseudo code

Step 1: Find the deletion node

```
p = findNode(x);   // Find the node that contains the value x
                   // ===> p is the "deletion node"
```

Step 2: Find the successor node in the RIGHT subtree of p

```
succ = p.right;    // Starting point: right subtree
while ( succ.left != null )
{
    succ = succ.left;    // Always go left to find min. value
}
```

Step 3: replace content of p with successor node

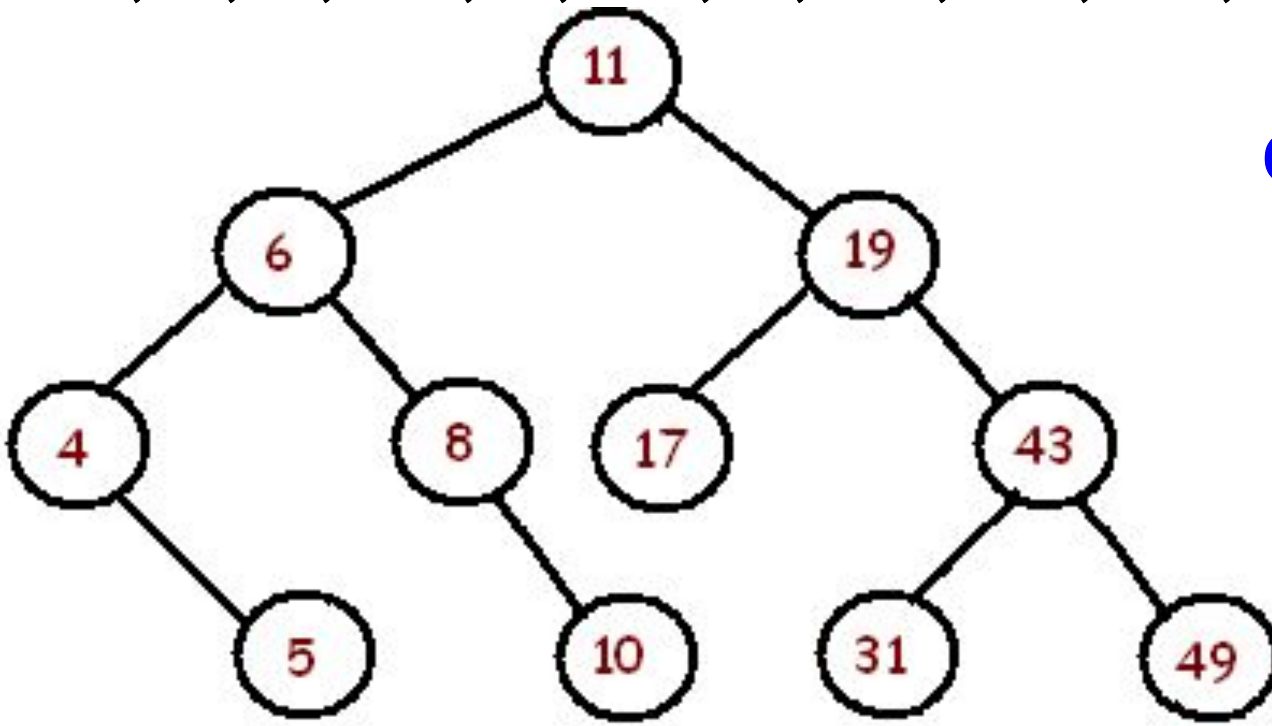
```
p.value = succ.value;
```

Step 4: delete successor node

```
succ's Parent.left = succ.right;    // Trouble: we need "succ's Parent"
```

Given binary search tree for a sequence of numbers:

11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 31



Observation ?

Pre-Order

In-Order

Post-Order

Pre-Order: 11, 6, 4, 5, 8, 10, 19, 17, 43, 31, 49

In-Order: 4, 5, 6, 8, 10, 11, 17, 19, 31, 43, 49 **(sorted)**

Post-Order: 5, 4, 10, 8, 6, 17, 31, 49, 43, 19, 11

THANK YOU

References:

http://www.mathcs.emory.edu/~cheung/Courses/171/Syllabus/9-BinTree/

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Trees/trees.html