DATA STRUCTURE

CHAPTER 7

TREE

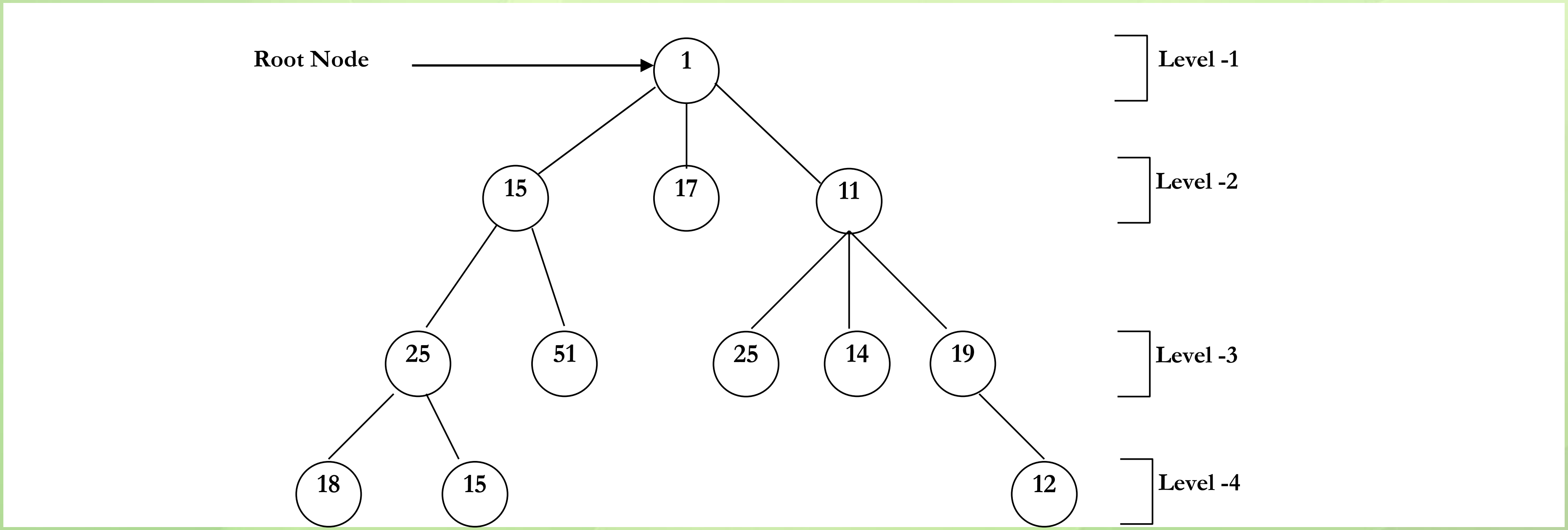
TREE

- We see tree in nature. The tree has root, branches, sub-branches and leaves. From the concept of natural tree, the computer scientists get the idea of a data structure, which is graphically similar to natural tree.
- Natural tree is a bottom-up figure. However, the graphical representation of the data structure tree is a top-down figure.

TREE

- A tree is a finite collection of nodes that has one to many relationship among nodes.
- A tree is a hierarchical structure.
- An ordered tree is a list of nodes that has a specially designated node called root node.
- The connection line between two nodes is called edge.
- The node that has no child node is called leaf node. The node that has child node is called parent node.
- A tree can be implemented (stored in memory) as an array or a linked list.

TREE



- Figure 7.1: A Tree

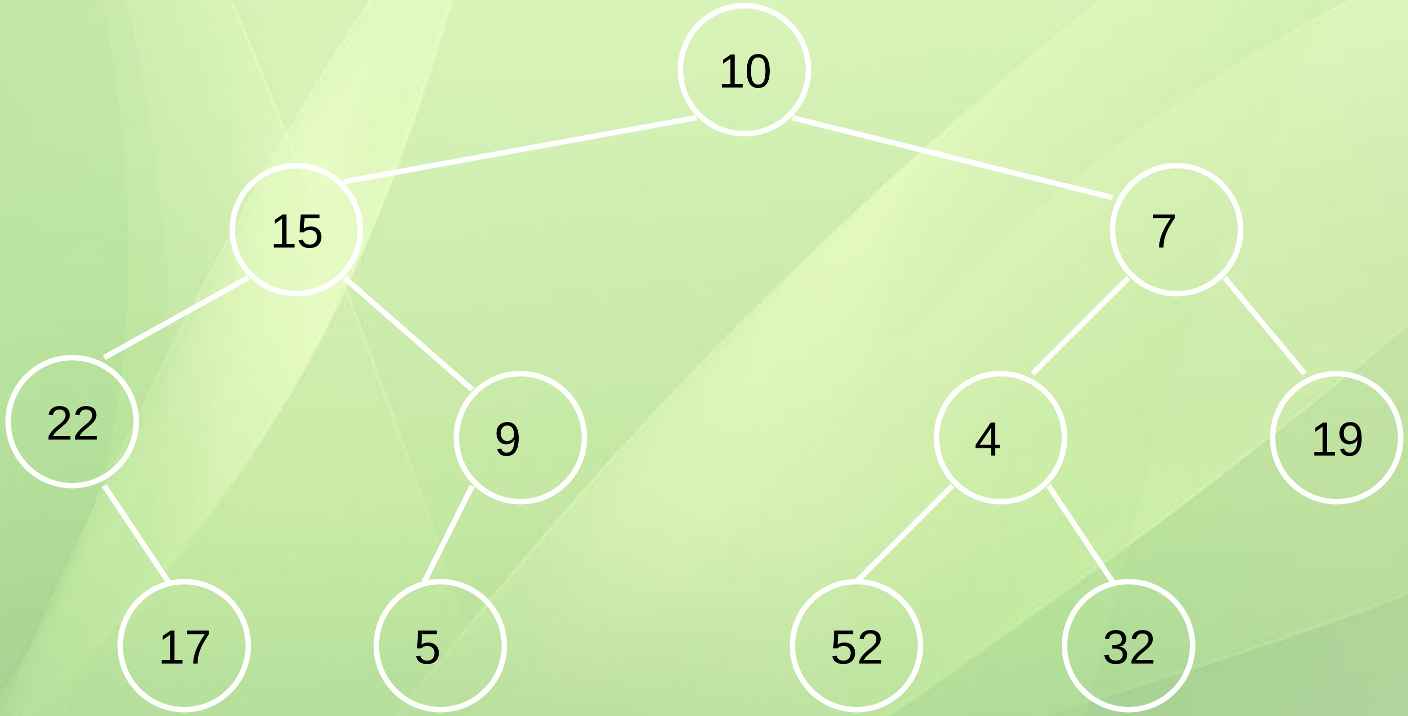
Binary Tree

- A binary tree is finite set of nodes, where there is a special node called root node and every node (including root node) has at best two children.

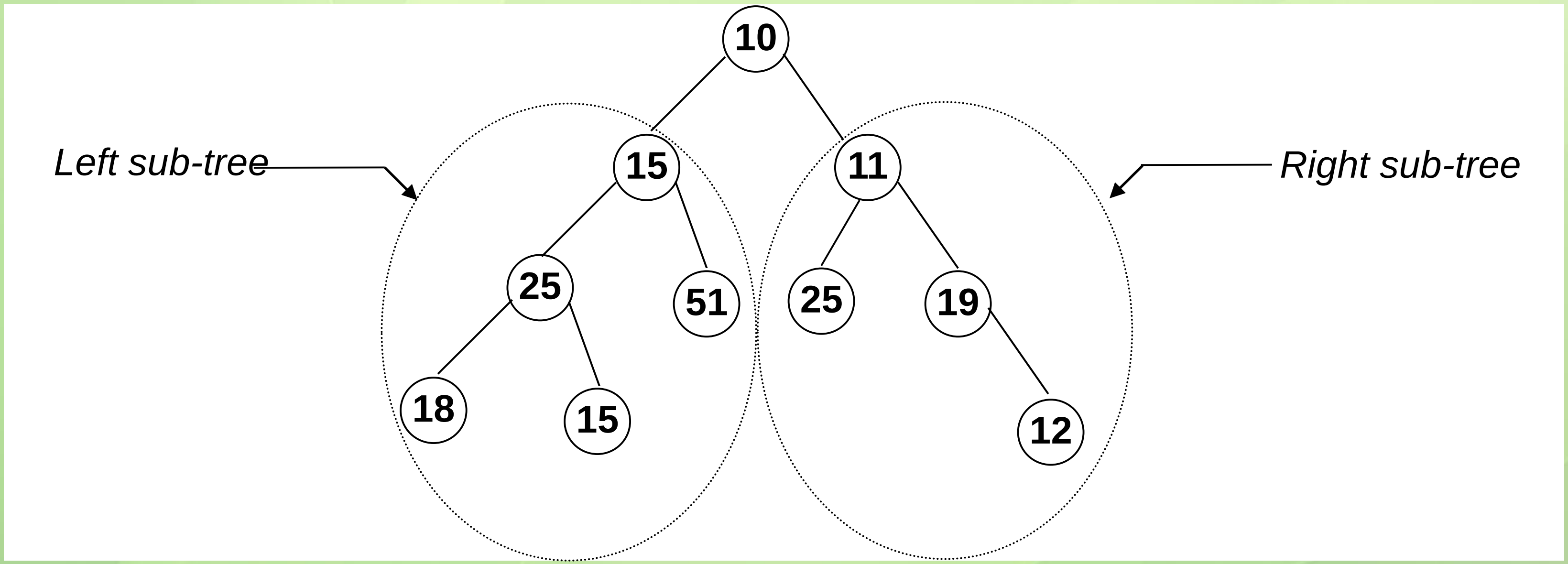
- The trees excluding root node are called left sub-tree and right sub-tree.

Binary Tree

- Each node can have at most 2 children

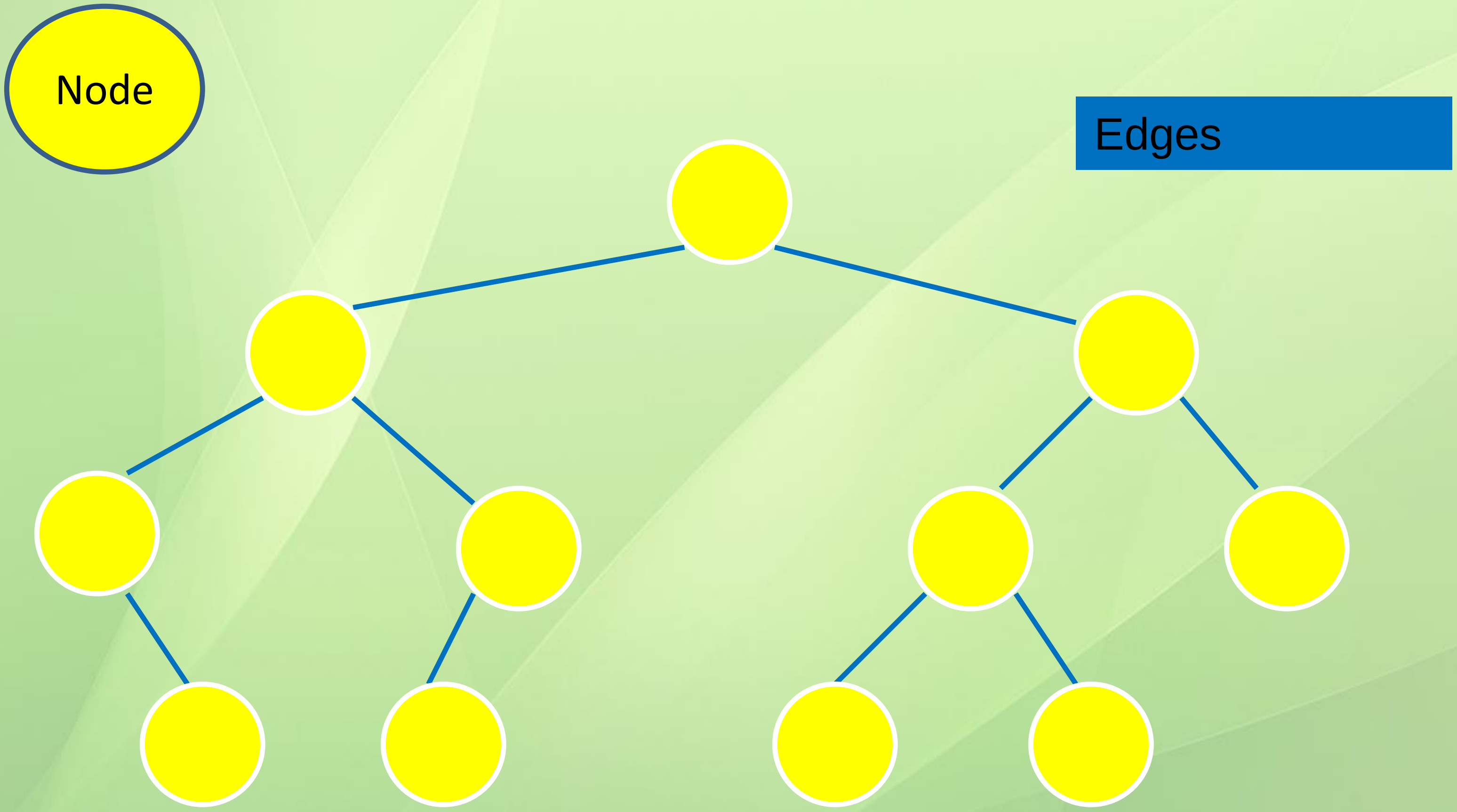


Binary Tree



- Figure 7.2: A binary tree

Parts of a Tree

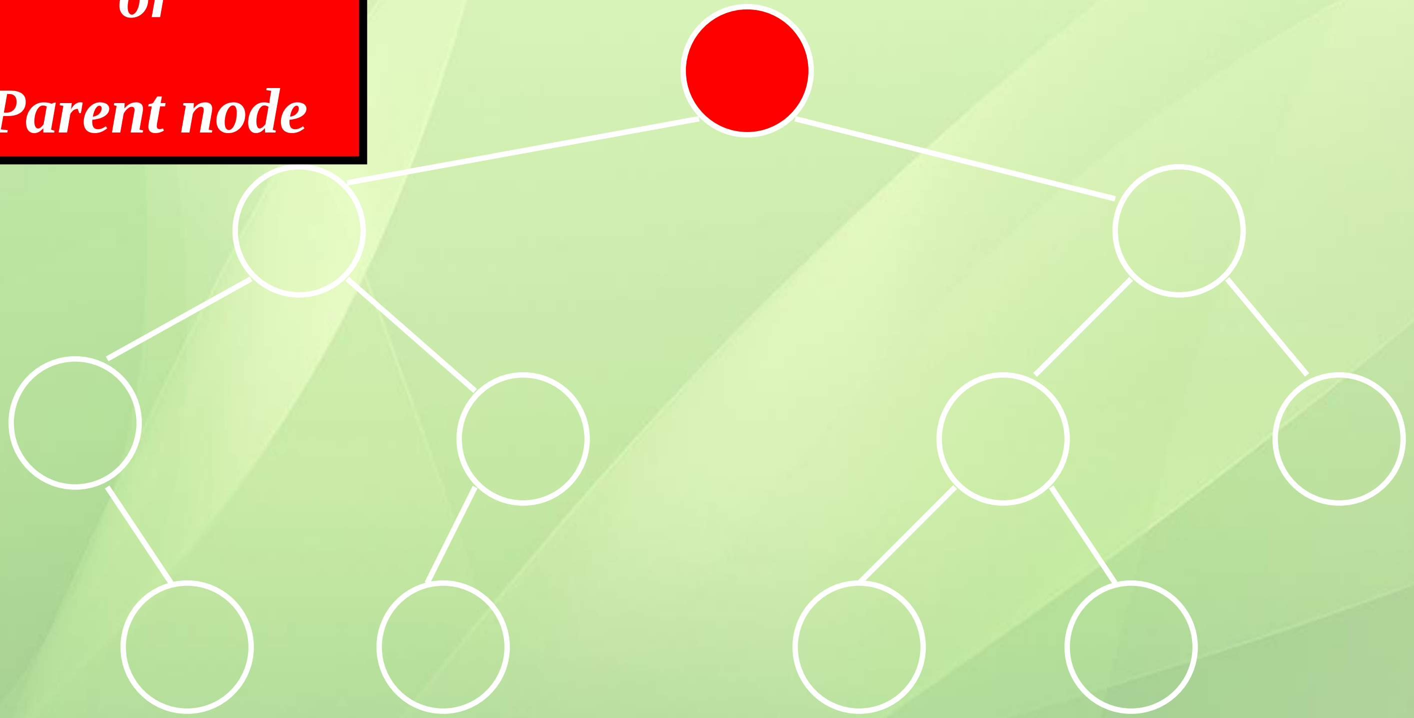


Parts of a Tree

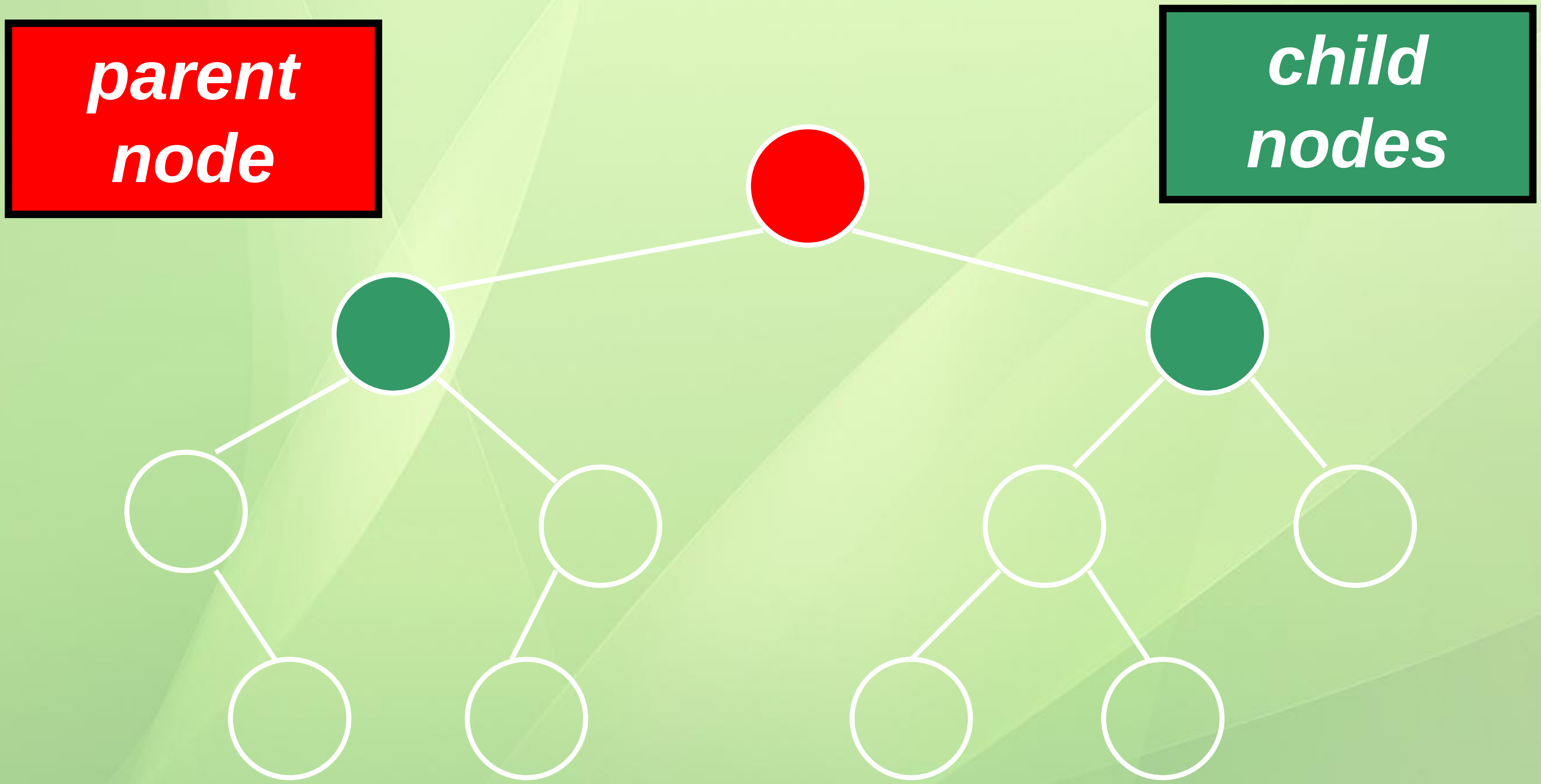
Root

or

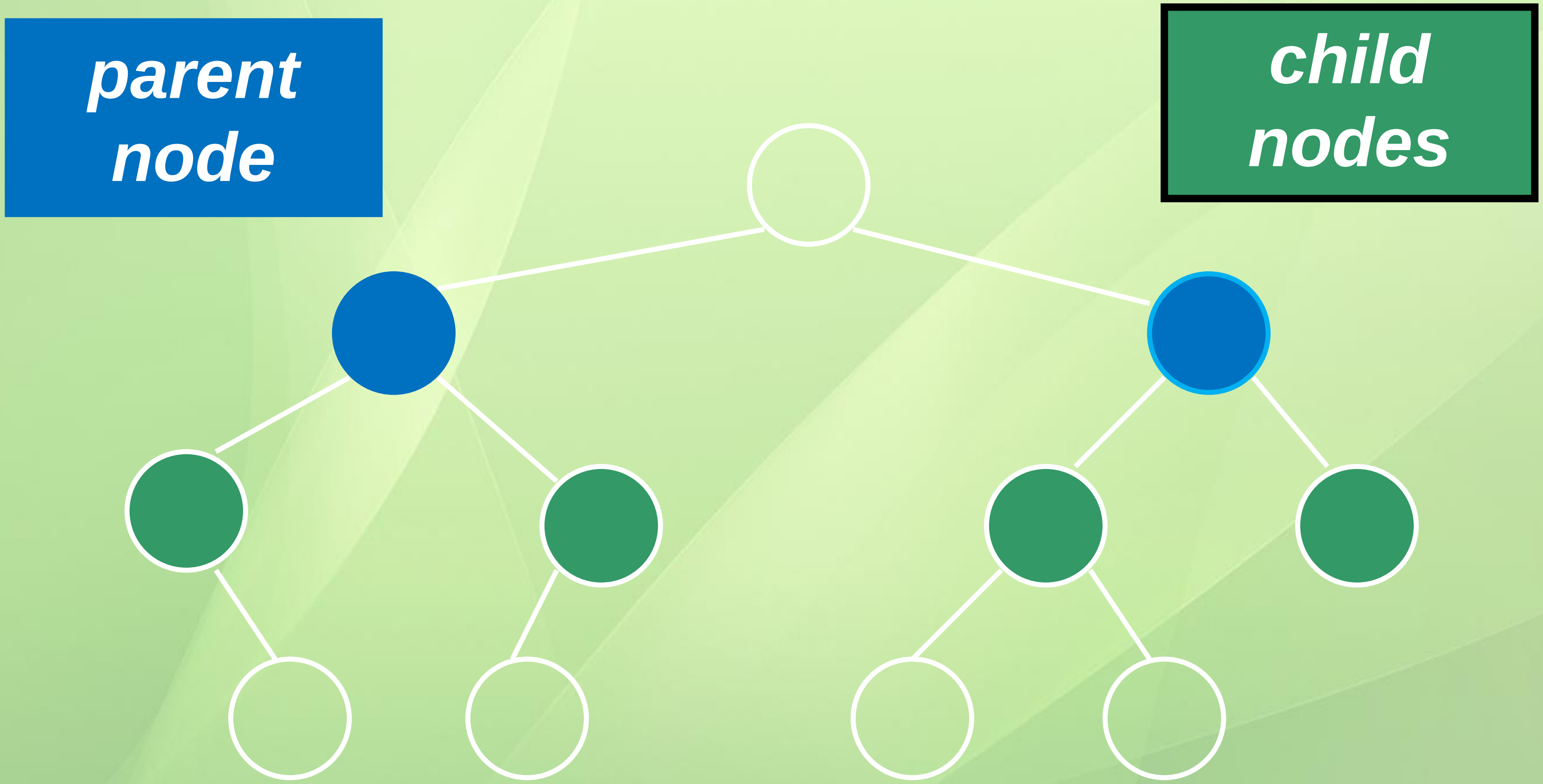
Parent node



Parts of a Tree

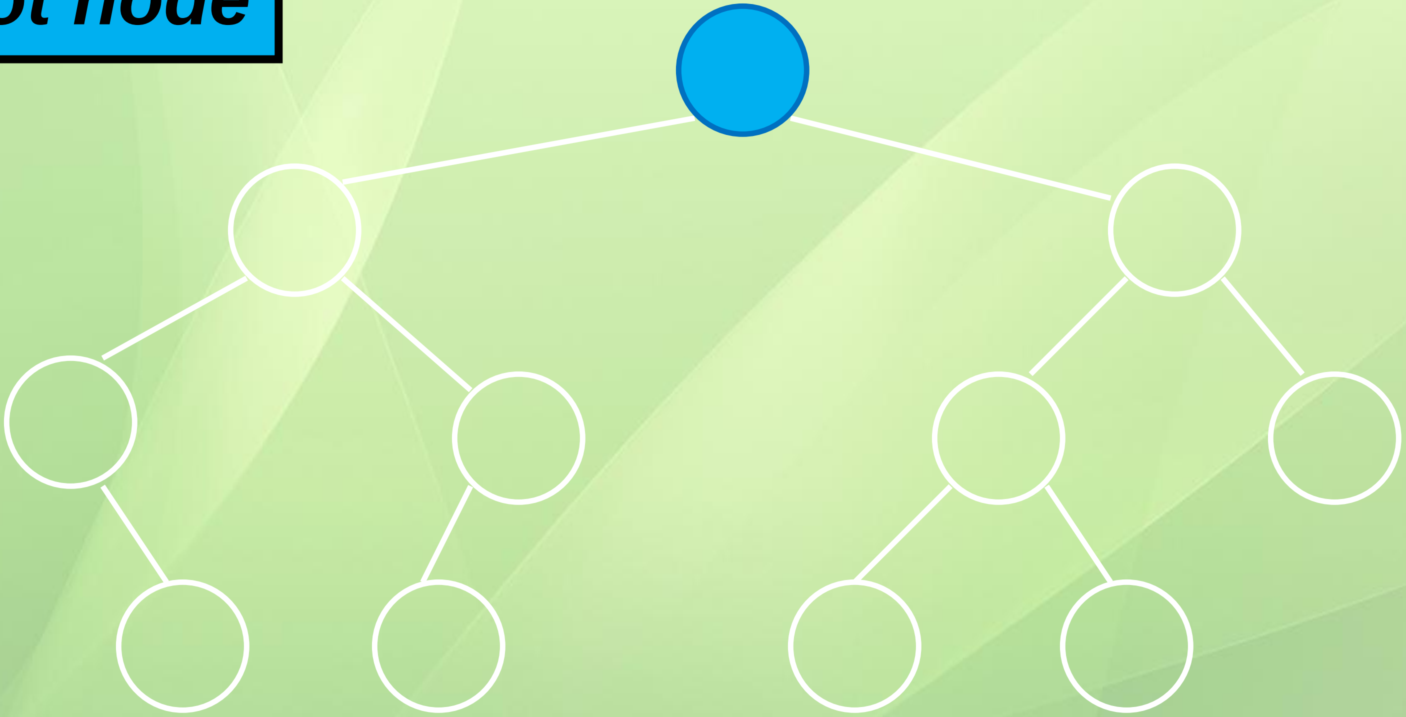


Parts of a Tree



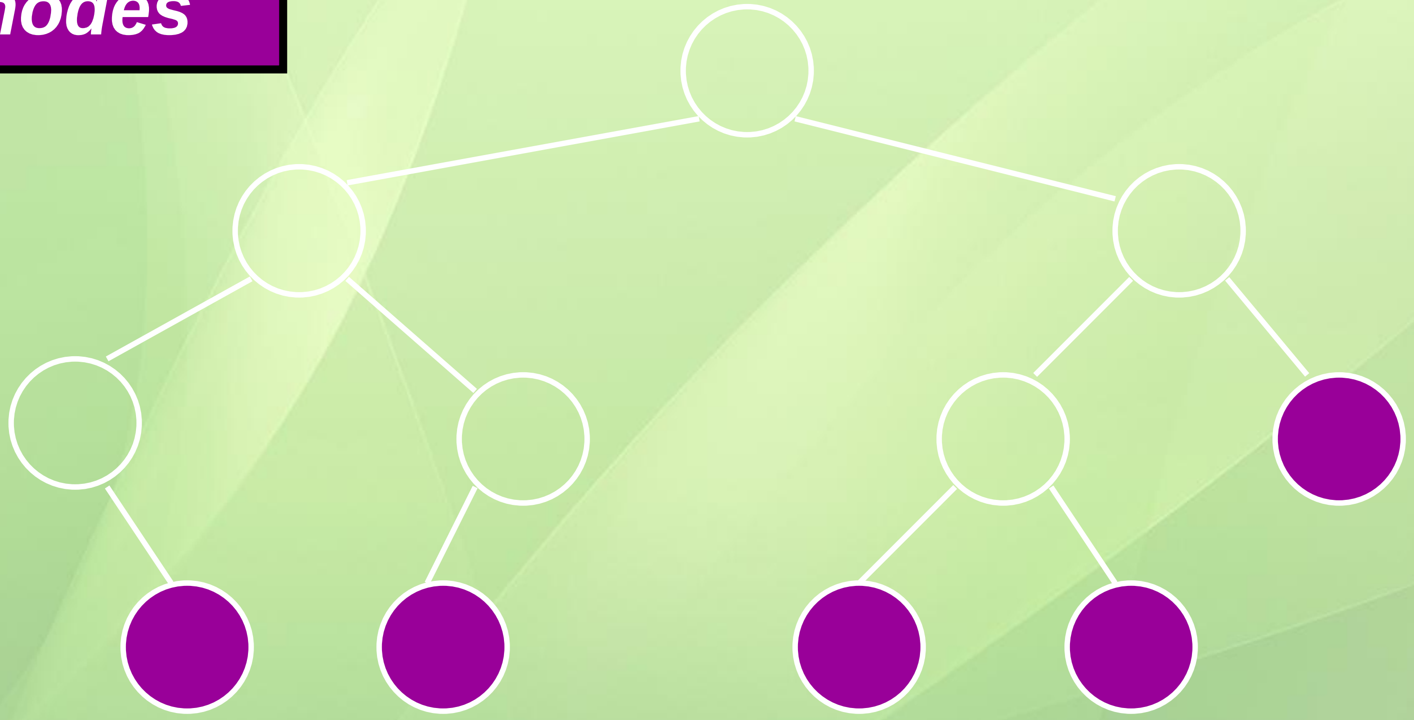
Parts of a Tree

root node



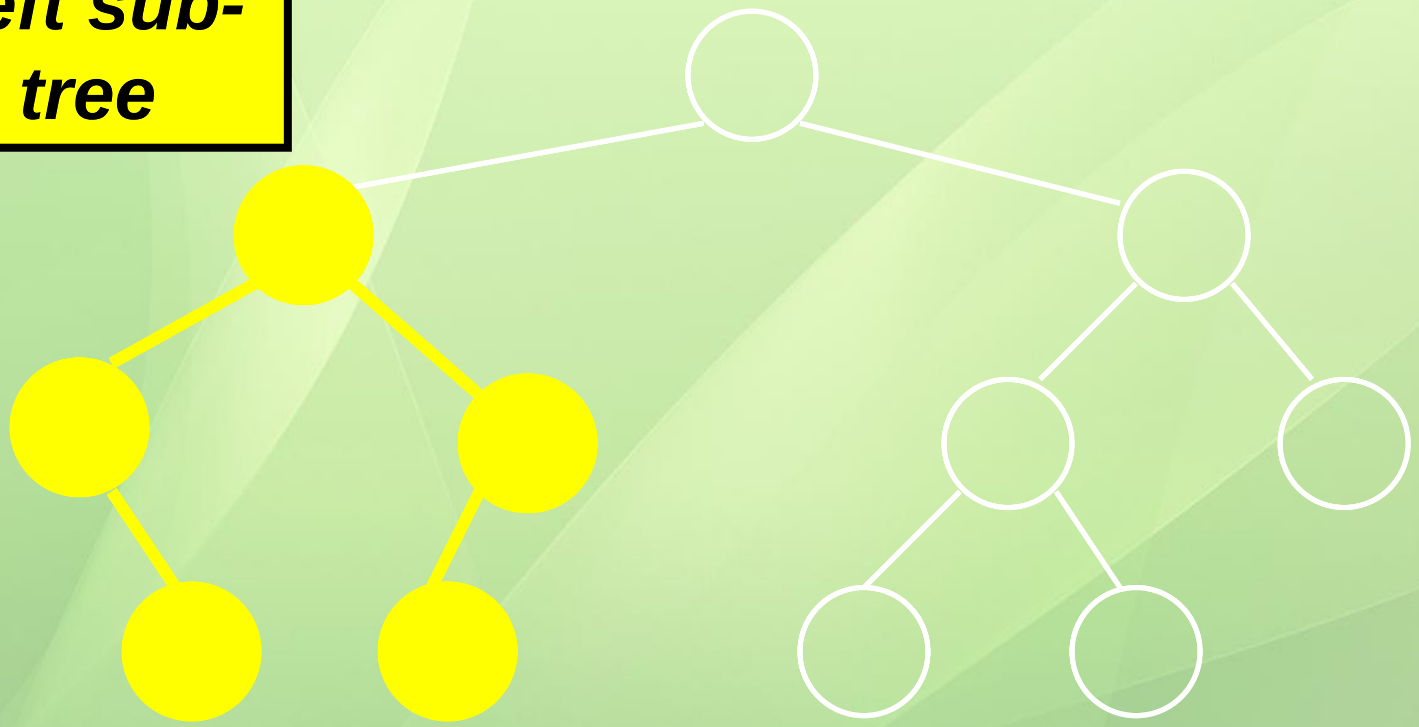
Parts of a Tree

leaf nodes



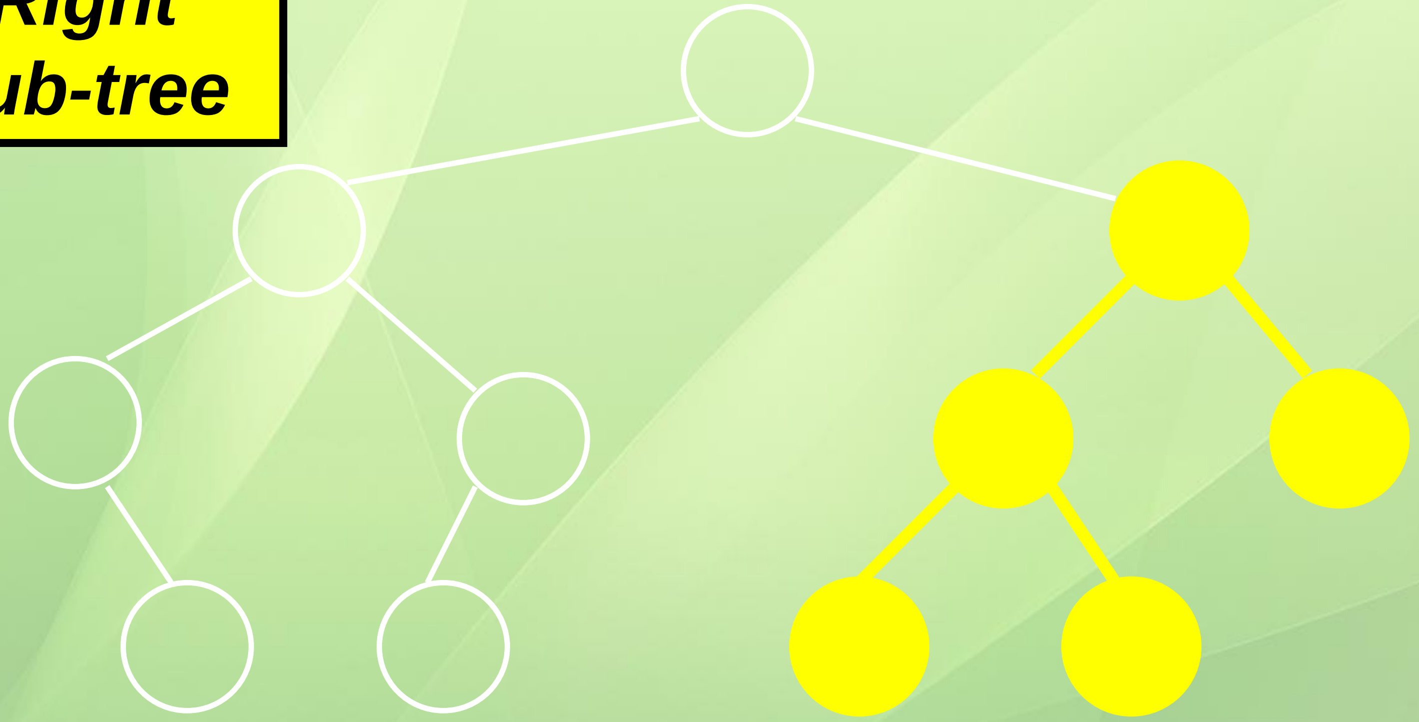
Parts of a Tree

Left sub-tree



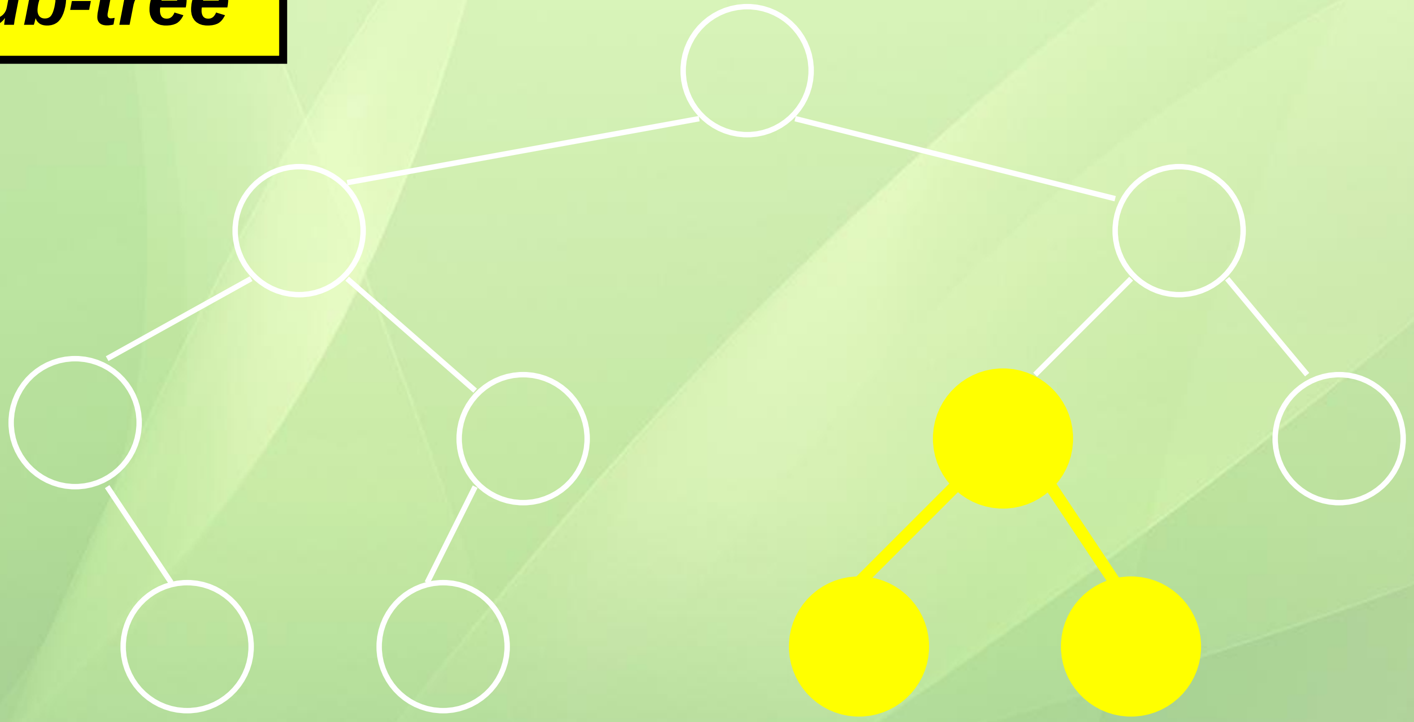
Parts of a Tree

Right sub-tree



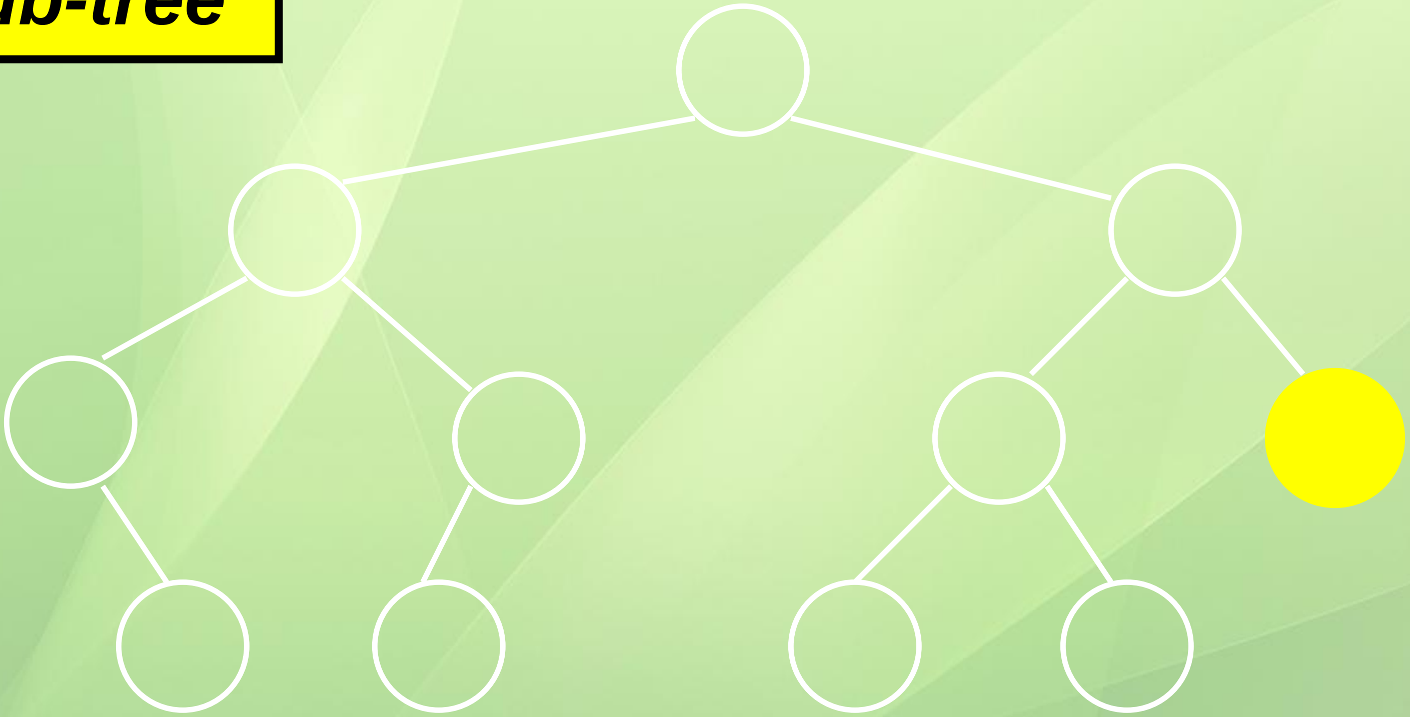
Parts of a Tree

sub-tree



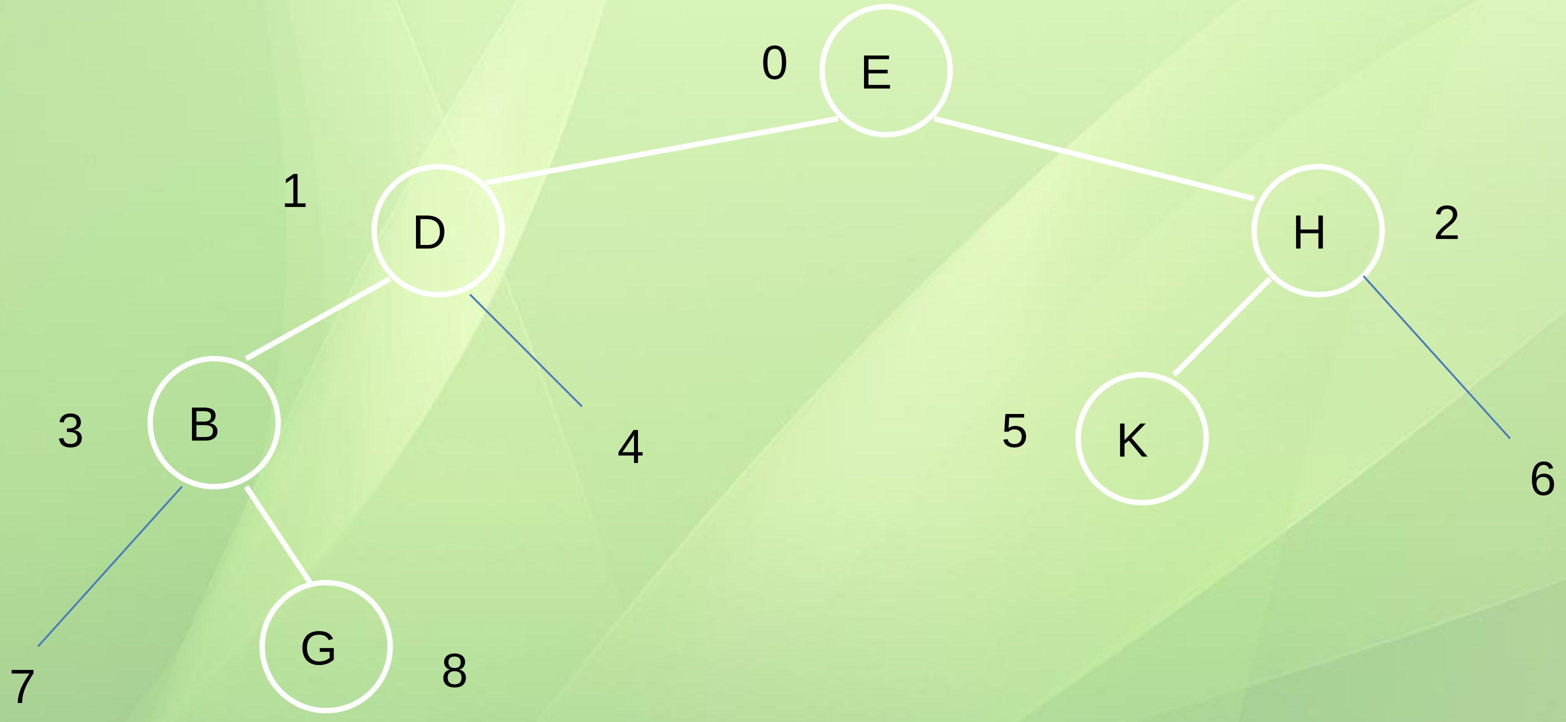
Parts of a Tree

sub-tree



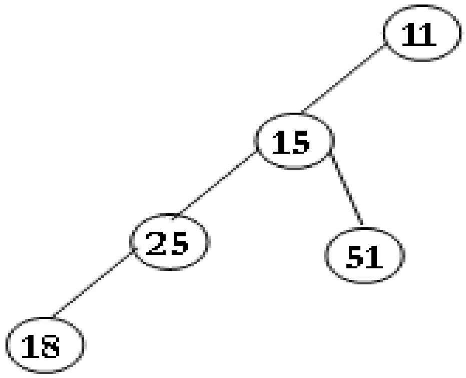
Array Representation of a Tree

0, 1, 2, 3, . . .are positions or indices

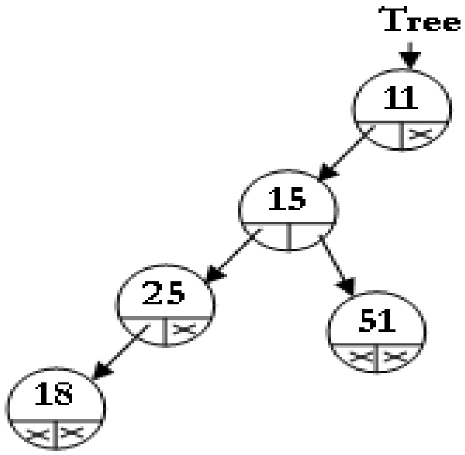


0	1	2	3	4	5	6	7	8
E	D	H	B		K			G

Binary Tree Representation



(a): A Binary Tree

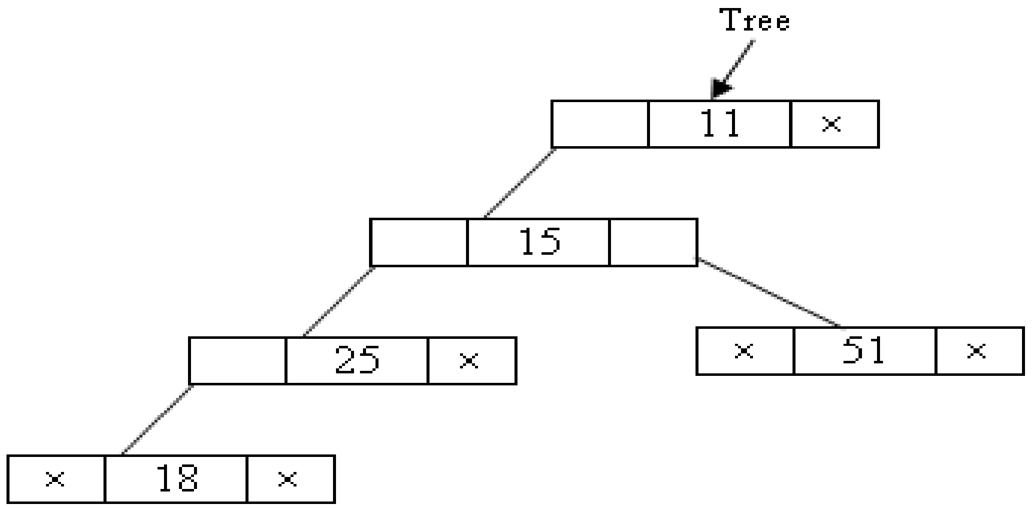


11	15	-	25	51			18

(b): Tree as an array

(c): Tree as a linked list (nodes are shown in round shape)

Binary Tree



(d) Tree as a linked list (nodes are shown in rectangular shape)

Figure 7.3: Tree implementation (store in memory)

Binary Tree

- ✓ **Full binary tree:**
  - If a binary tree contains nodes in such a way that every node has at least two children except the leaf nodes, then the tree is called a full binary tree.
- ✓ **Complete binary tree:**
  - If a binary tree contains nodes in such a way that every level except the deepest(last) has as many nodes as possible and the nodes of the deepest level are in as left as possible, the tree is called complete binary tree.(online picture)
  - All full binary trees are complete binary trees.

Binary Tree

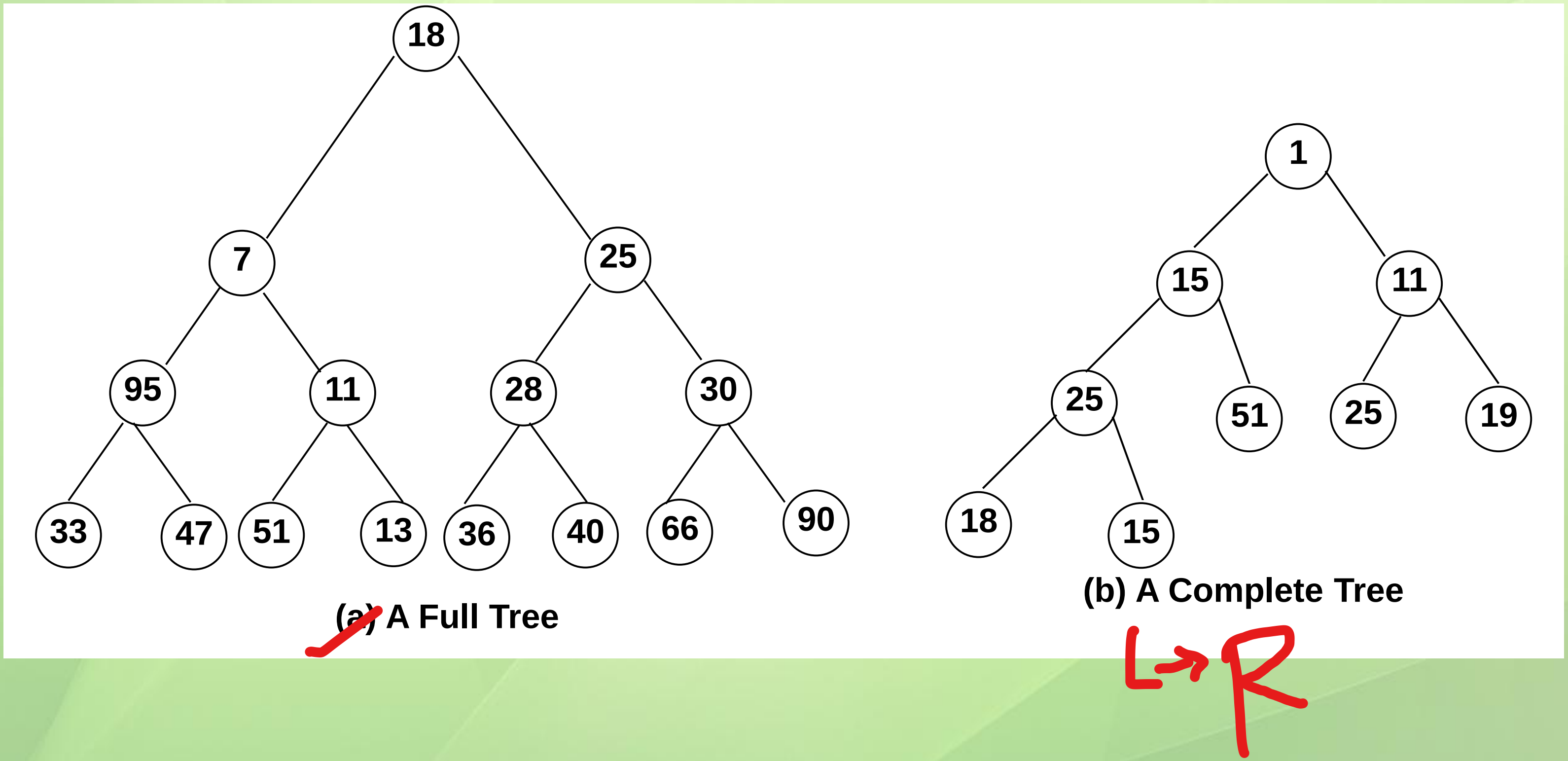


Figure 7.4: Pictorial view of Full tree and complete tree

Binary Tree

If there are k levels in a **full binary tree**, then the number of nodes is as follows:

$$n = 2^k - 1.$$

node $= 2^k - 1$

For example, if $k = 3, n = 7$

$k = 4, n = 15.$

When n is known, then for **complete or full binary** tree we get,

$$k = \lceil \log_2 (\underline{n+1}) \rceil$$ // $\lceil x \rceil$ means ceiling of x to the next integer

For example,

when $n = 17$; $k = \lceil \log_2(17+1) \rceil = \lceil \log_2(18) \rceil = \lceil 4.1...... \rceil = 5$

when $n = 34$; $k = \lceil \log_2(34+1) \rceil = \lceil \log_2(35) \rceil = \lceil 5.2.... \rceil = 6$; etc.

Binary Tree in **Array**

Parent child relationship(from parent to child):

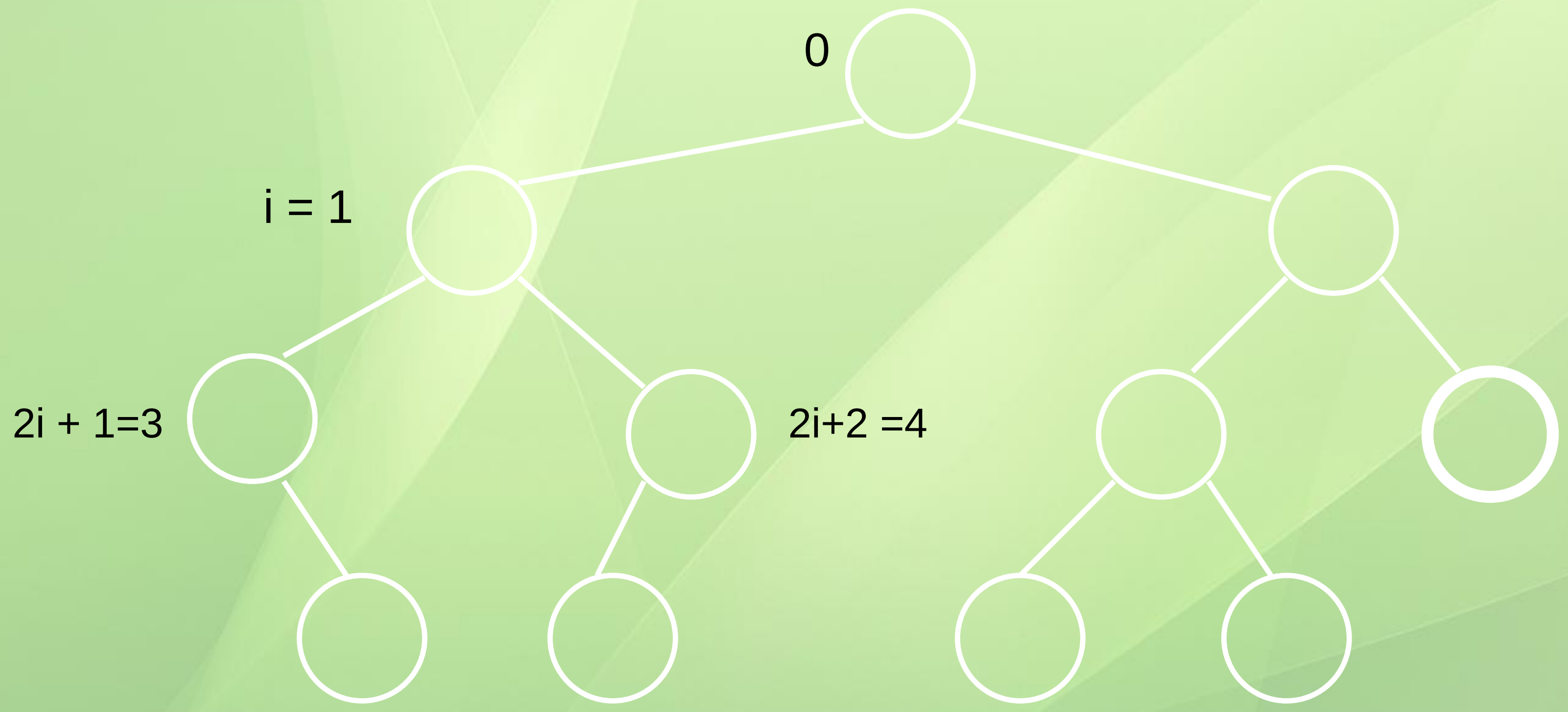
If we consider the root's **position is *0***,

Parent's position(index) = i,

Left child's position(index) = 2i+1,

Right child's position(index) = 2i+2.

Parent Child Relationship



Binary Tree in Array

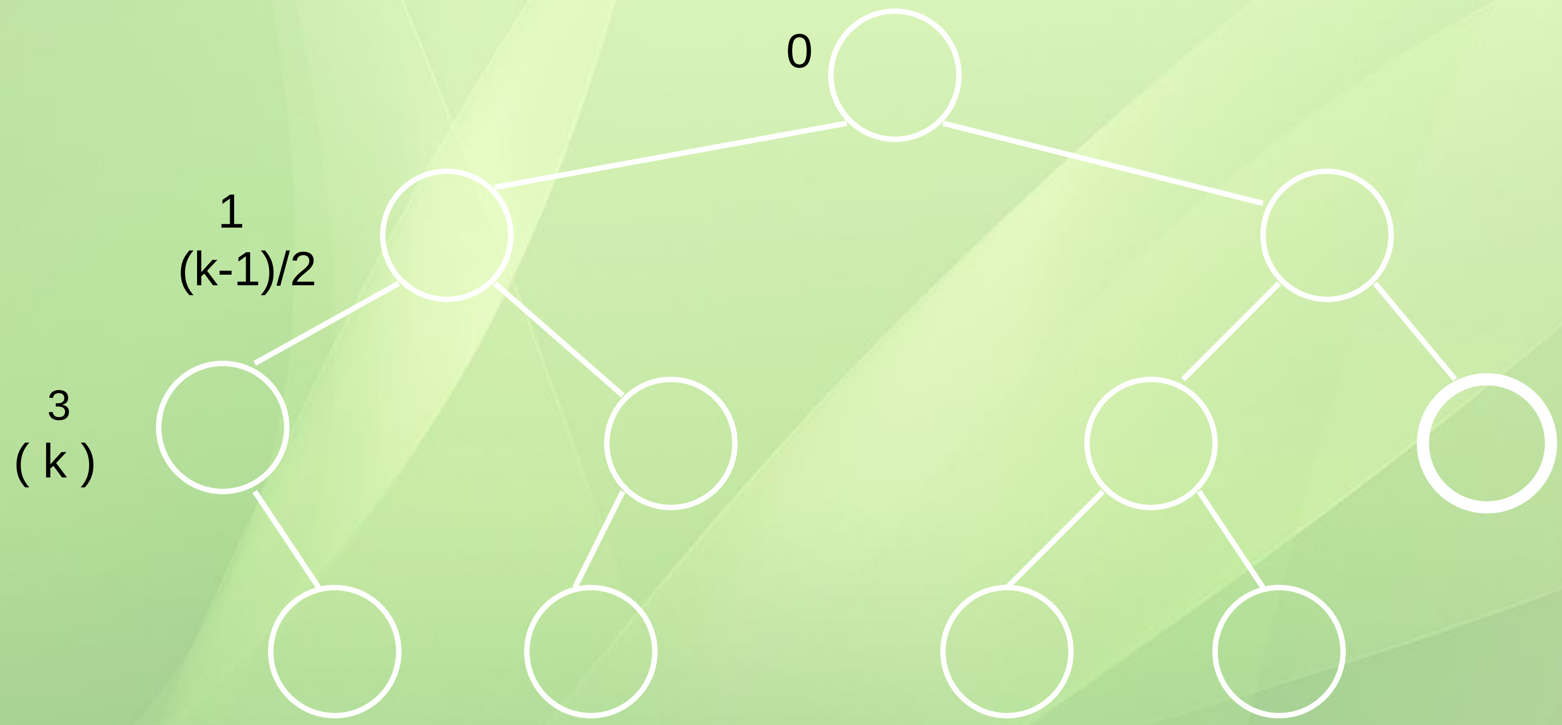
Child parent relationship (from child to parent):

Child's position (index): k

Parent's position: (k-1)/2,

Where / denotes Integer division.

Child Parent Relationship



Binary Tree in **Array**

Parent child relationship(from parent to child):

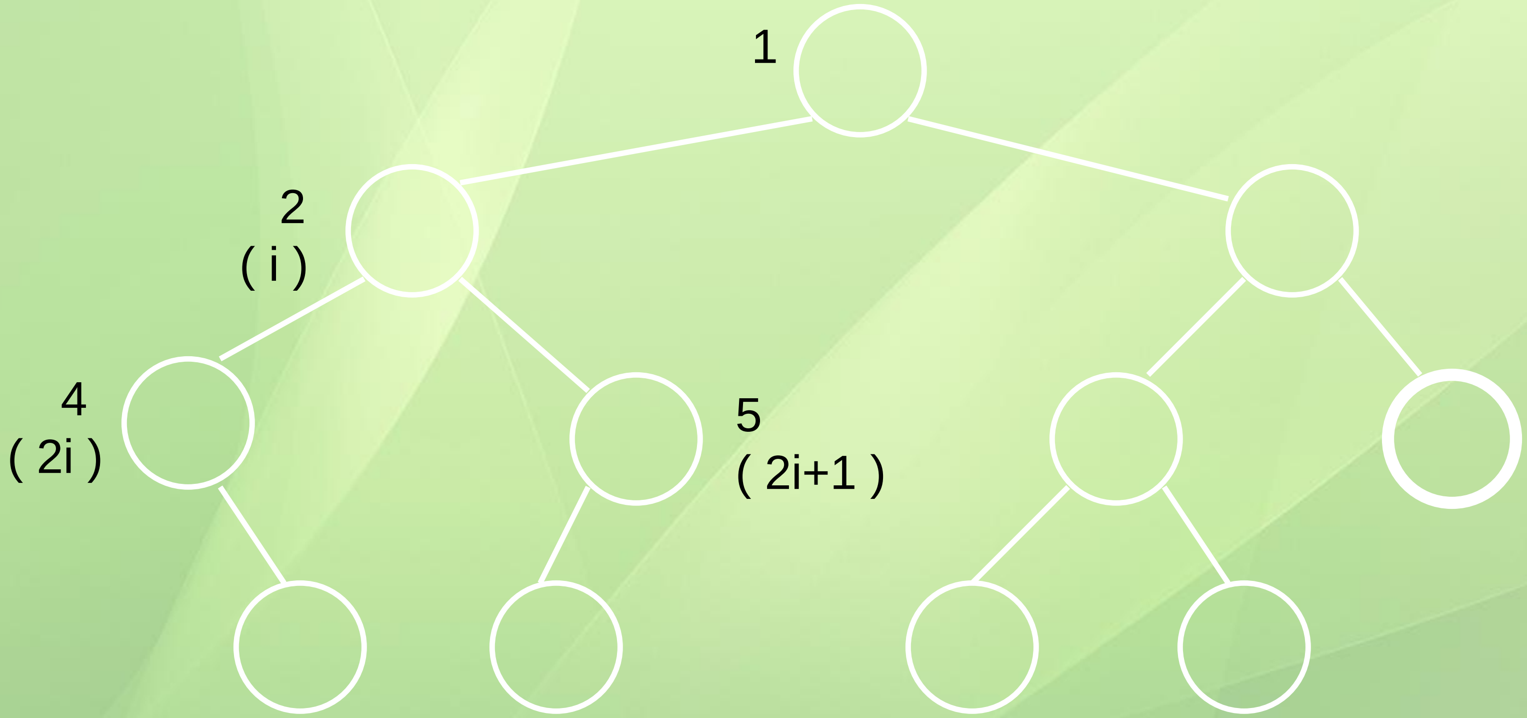
If we consider the root's **position is 1**,

Parent's position(index) = i,

Left child's position(index) = 2i,

Right child's position(index) = 2i + 1.

Parent Child Relationship



Binary Tree in Array

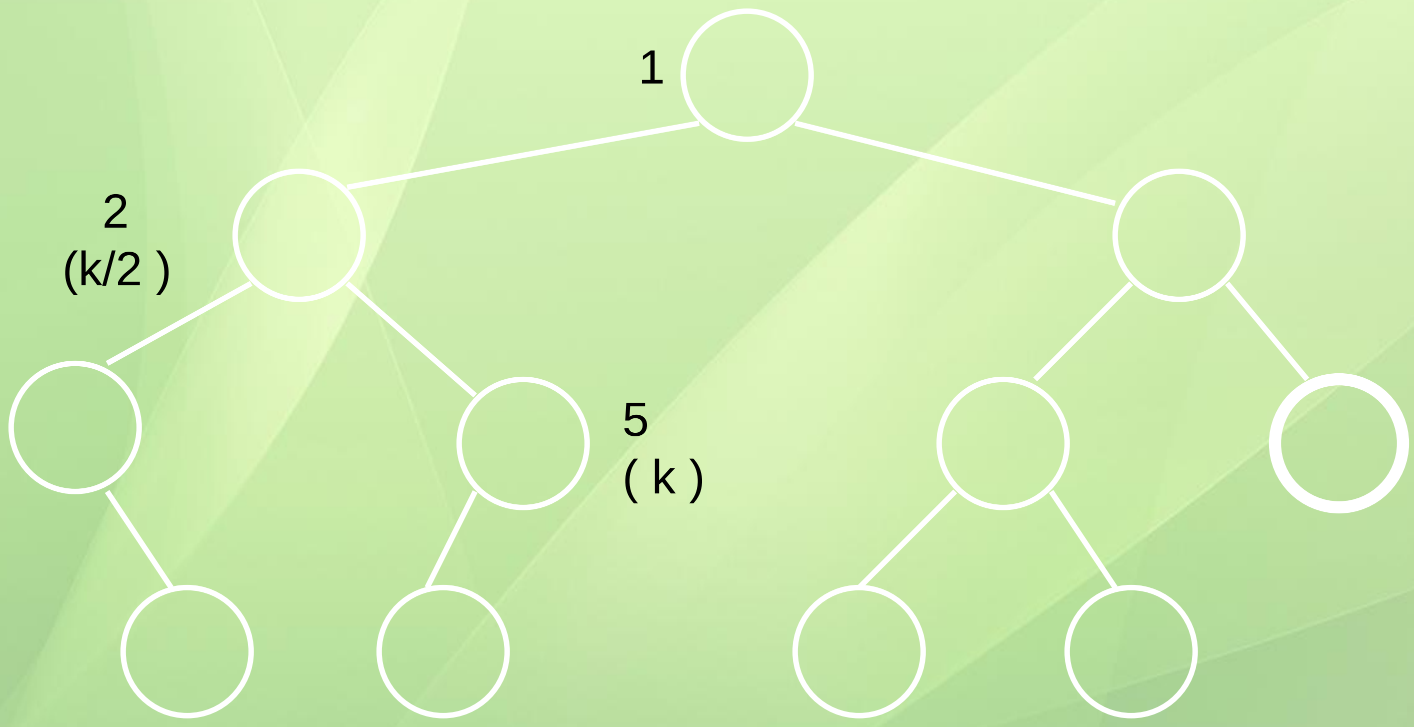
Child parent relationship (from child to parent):

Child’s position (index): k

Parent’s position: k/2, / denotes Integer division.

If the binary tree is a complete then array representation is efficient.

Child Parent Relationship



Linked representation of Binary Tree

```
struct node
   {
   int data;
   node *lchild;   //pointer to left child
   node *rchild;   //pointer to right child
    };
```

If the tree is not complete then linked representation is efficient.

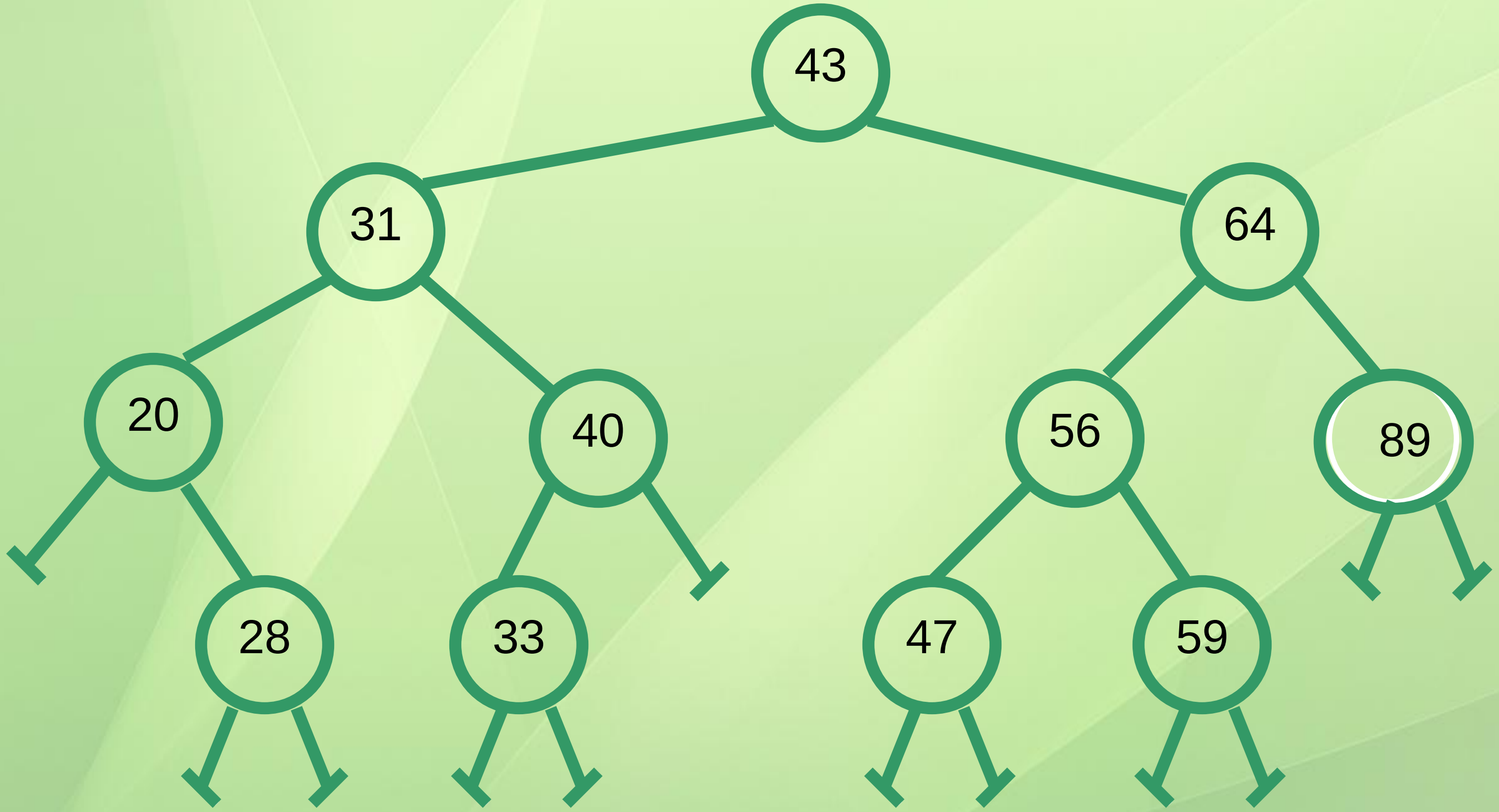
Traversal Technique of a Binary Tree

- There are three main traversal techniques (methods) for a binary tree. Such as
  - Pre-order Traversal Method
  - In-order Traversal Method
  - Post-order Traversal Method

Pre-order Traversal Method

- Visit the root (node).
- Traverse the left sub-tree (in pre-order)
- Traverse the right sub-tree (in pre-order)

Example: Preorder



43	31	20	28	40	33	64	56	47	59	89

Another Example of Preorder

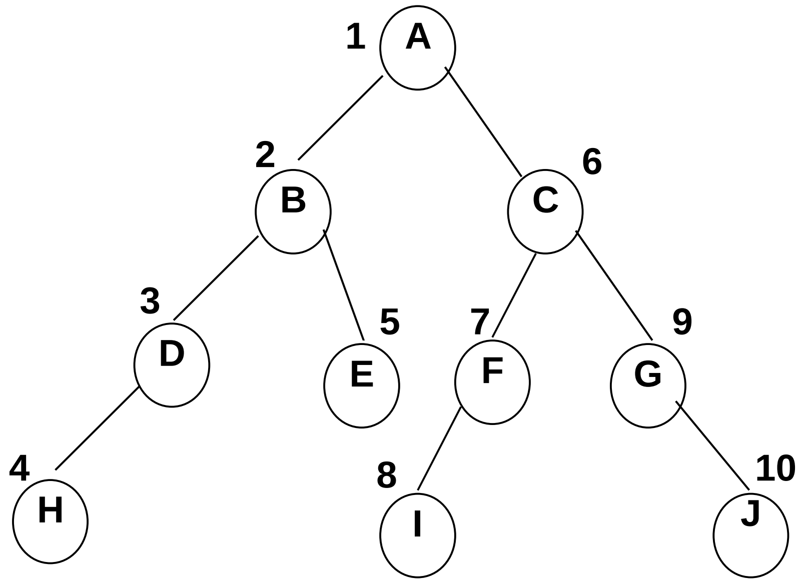


Figure 7.5: Pre-order traversal method

Visiting sequence:

A B D H E C F I G J

✓ Pre-order Traversal Method

Algorithm 7.1: Algorithm for pre-order traversal

```
1. Input a binary tree
2. preorder (node * curptr)
{
   if (curptr!= NULL)
      {
       print curptr→data;
       preorder (curptr→lchild);
       preorder (curptr→ rchild);
       }
}
3. Output: the information of the nodes.
```

Array based algorithm for preorder method

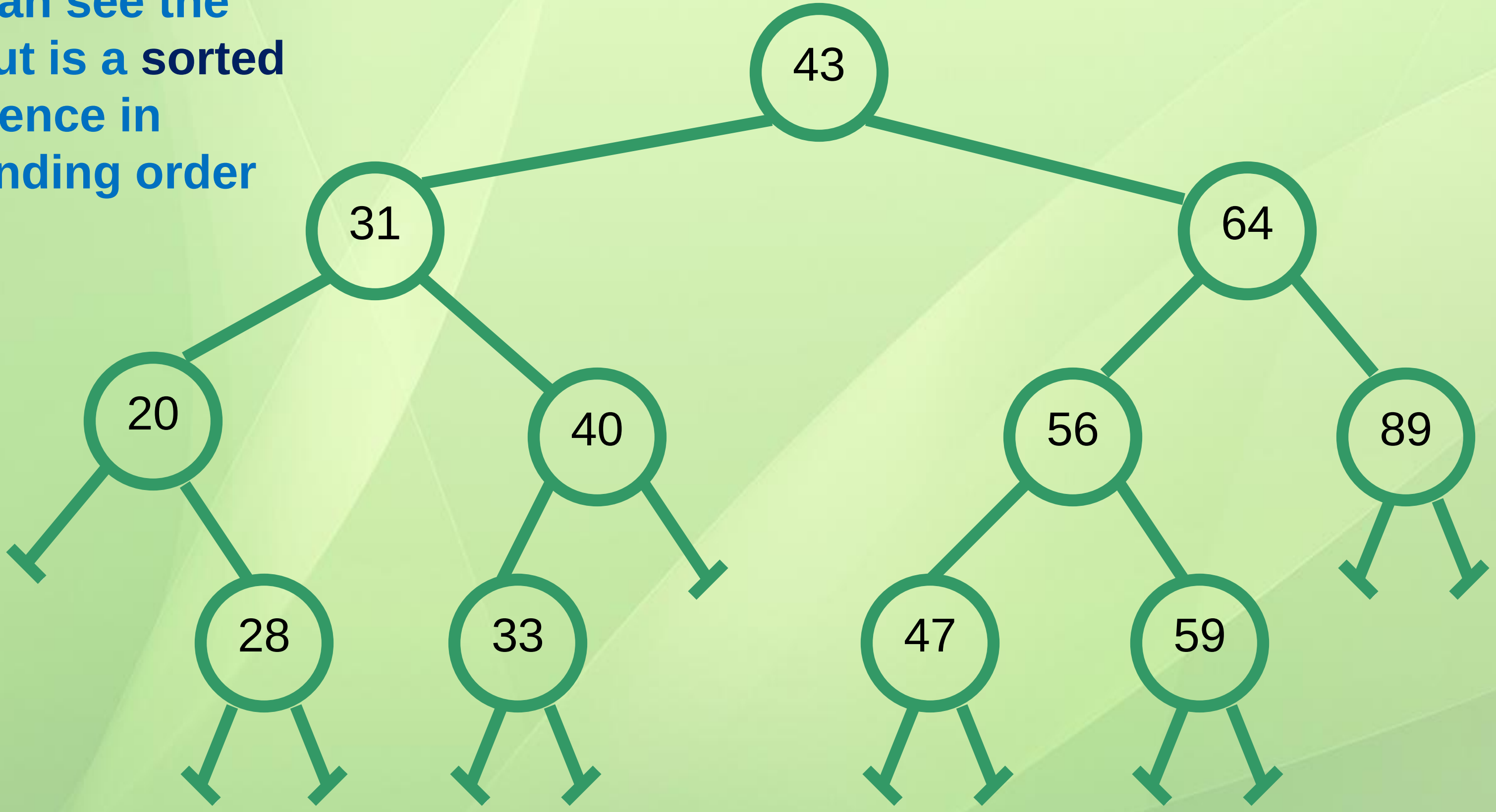
```
preorder (int i, int tree [ ])
{
if (i<n) cout<< tree[i];
preorder (2*i+1, tree);
preorder(2*i+2, tree);
main()
{
input tree[n]// an array of size n
preorder (0, tree);
}
```

Inorder Traversal

- ❑Traverse the left subtree (inorder).
- ❑Visit the root (node).
- ❑Traverse the right subtree (inorder).

Example: Inorder

We can see the output is a **sorted** sequence in ascending order



20	28	31	33	40	43	47	56	59	64	89

In-order Traversal

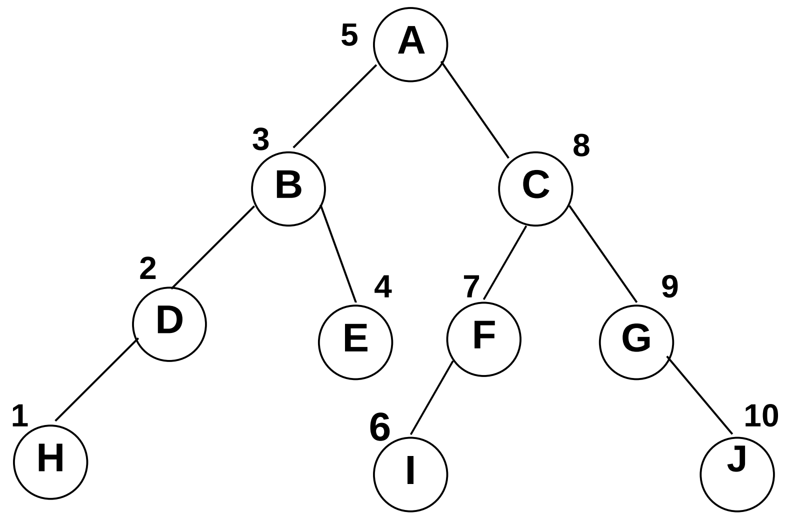


Fig-(a)

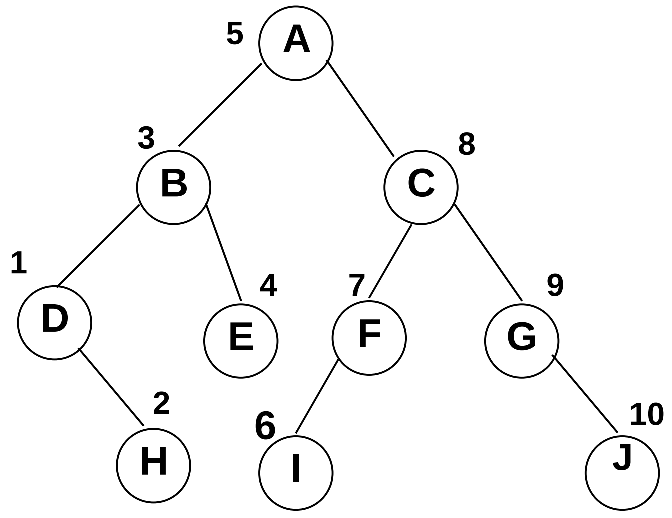


Fig-(b)

Visiting Sequences:

H D B E A I F C G J

D H B E A I F C G J

In-order Traversal Method

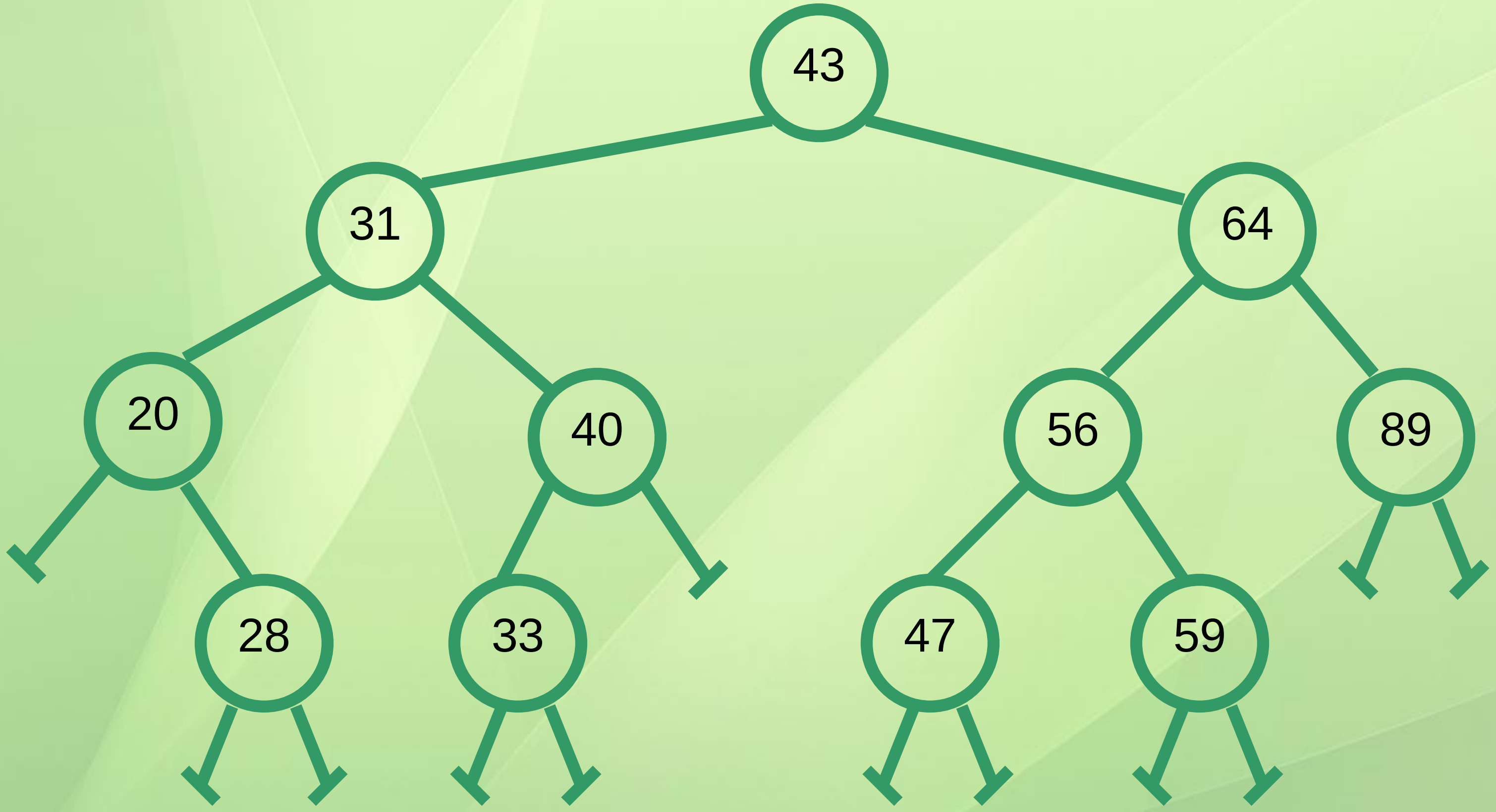
Algorithm 7.2: Algorithm for in-order traversal

```
1. Input a binary tree.
2. inorder (node * curptr)
{
      if (curptr!= NULL)
      {
            inorder (curptr→lchild);
            print curptr→data;
            inorder (curptr→rchild);
      }
}
3. Output: the information of the nodes.
```

Postorder Traversal

- ❑Traverse the left subtree (postorder).
- ❑Traverse the right subtree (postorder).
- ❑Visit the root (node).

Example: Postorder



28	20	33	40	31	47	59	56	89	64	43

Post-order Traversal Method

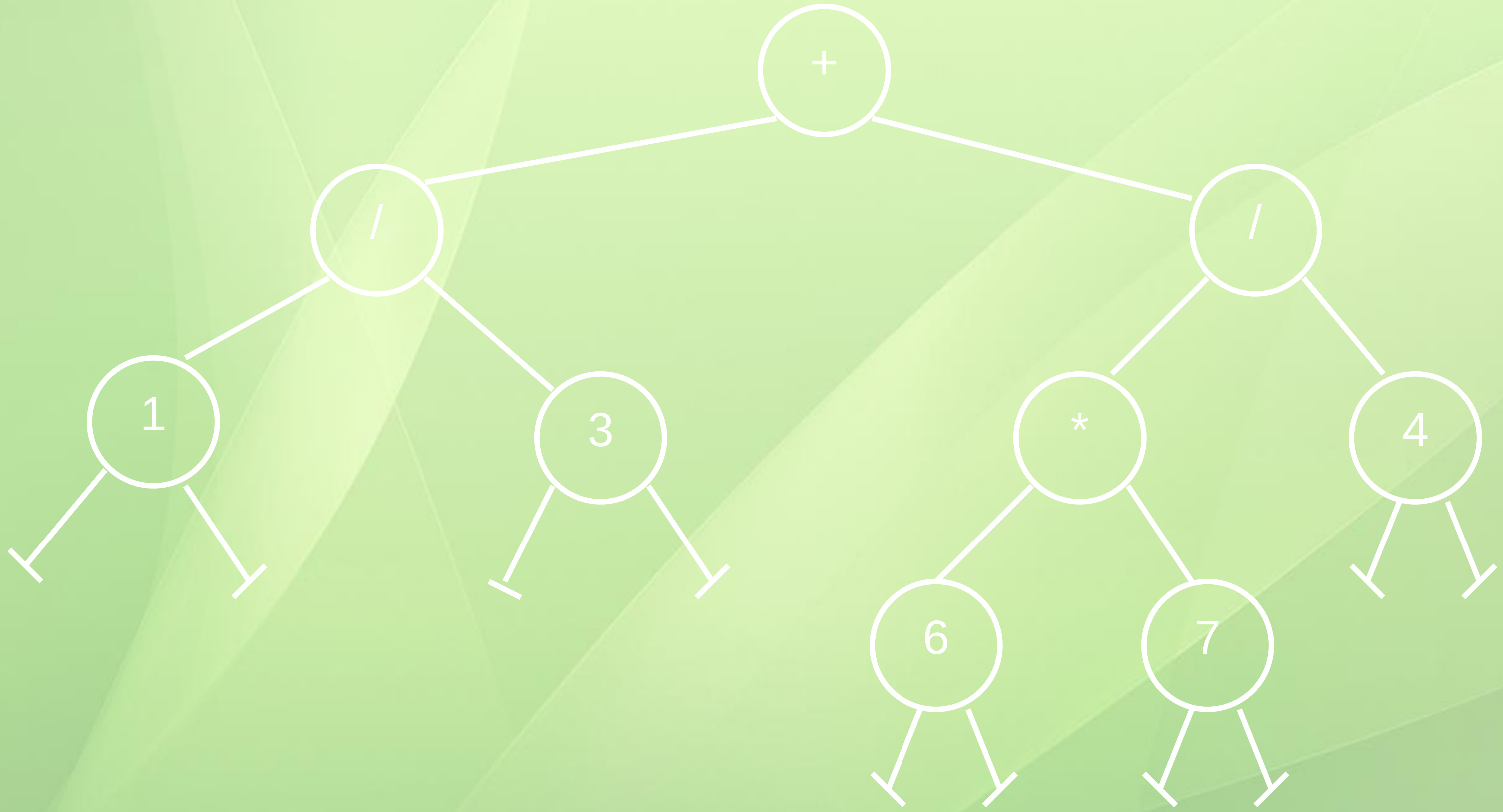
Algorithm 7.3: Algorithm for post-order traversal

```
1.Input a binary tree.
2. postorder (node * curptr)
{
if (curptr!= NULL)
        {
        postorder (curptr→lchild);
        postorder (curptr→rchild);
        print curptr→data;
        }
}
3. Output: the information of the nodes.
```

Expression Tree

- A Binary Tree built with operands and operators.
- Also known as a parse tree.
- Use in compilers.

Example: Expression Tree



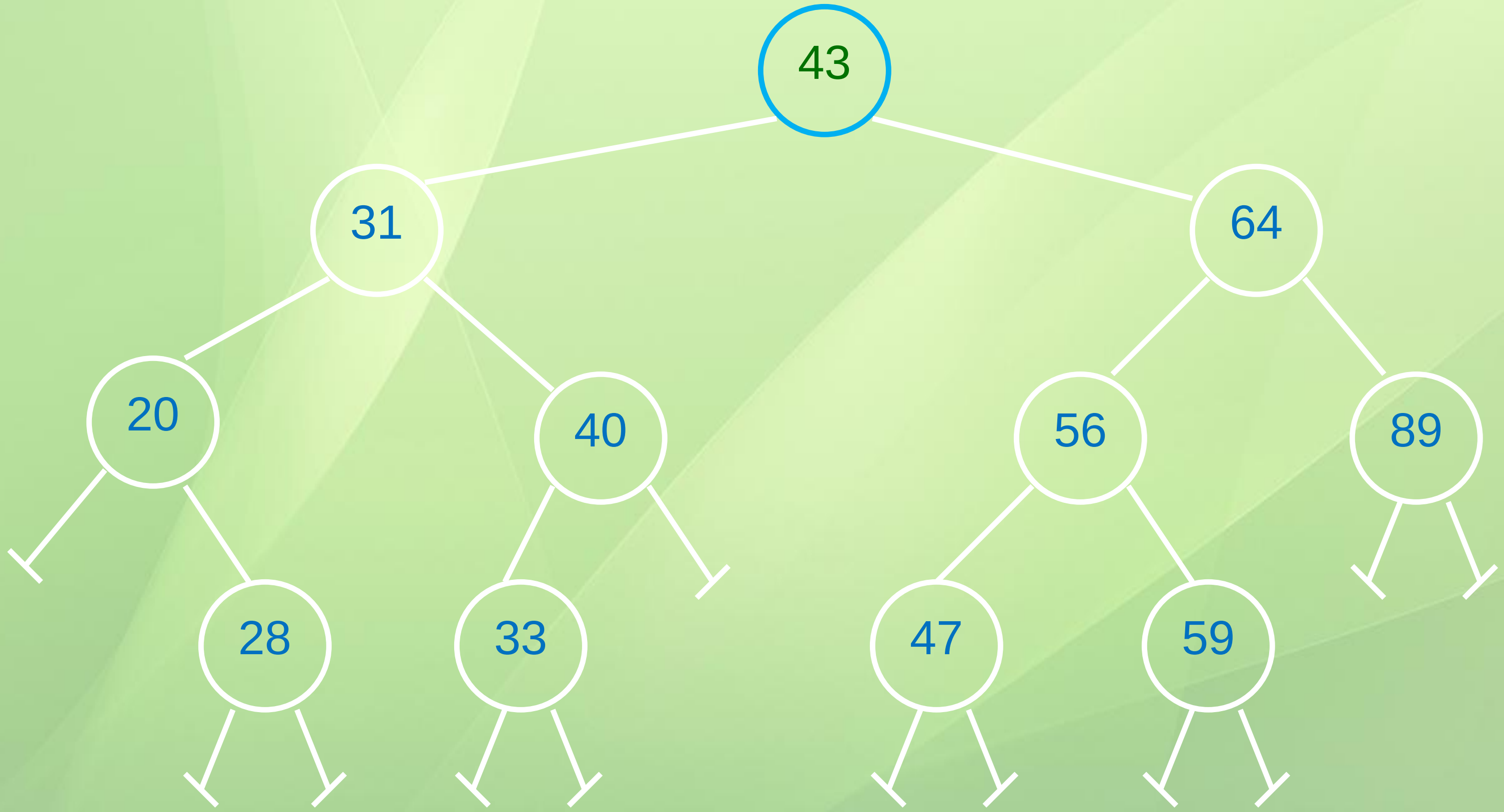
1/3 + 6*7 / 4

Binary Search Tree(BST)

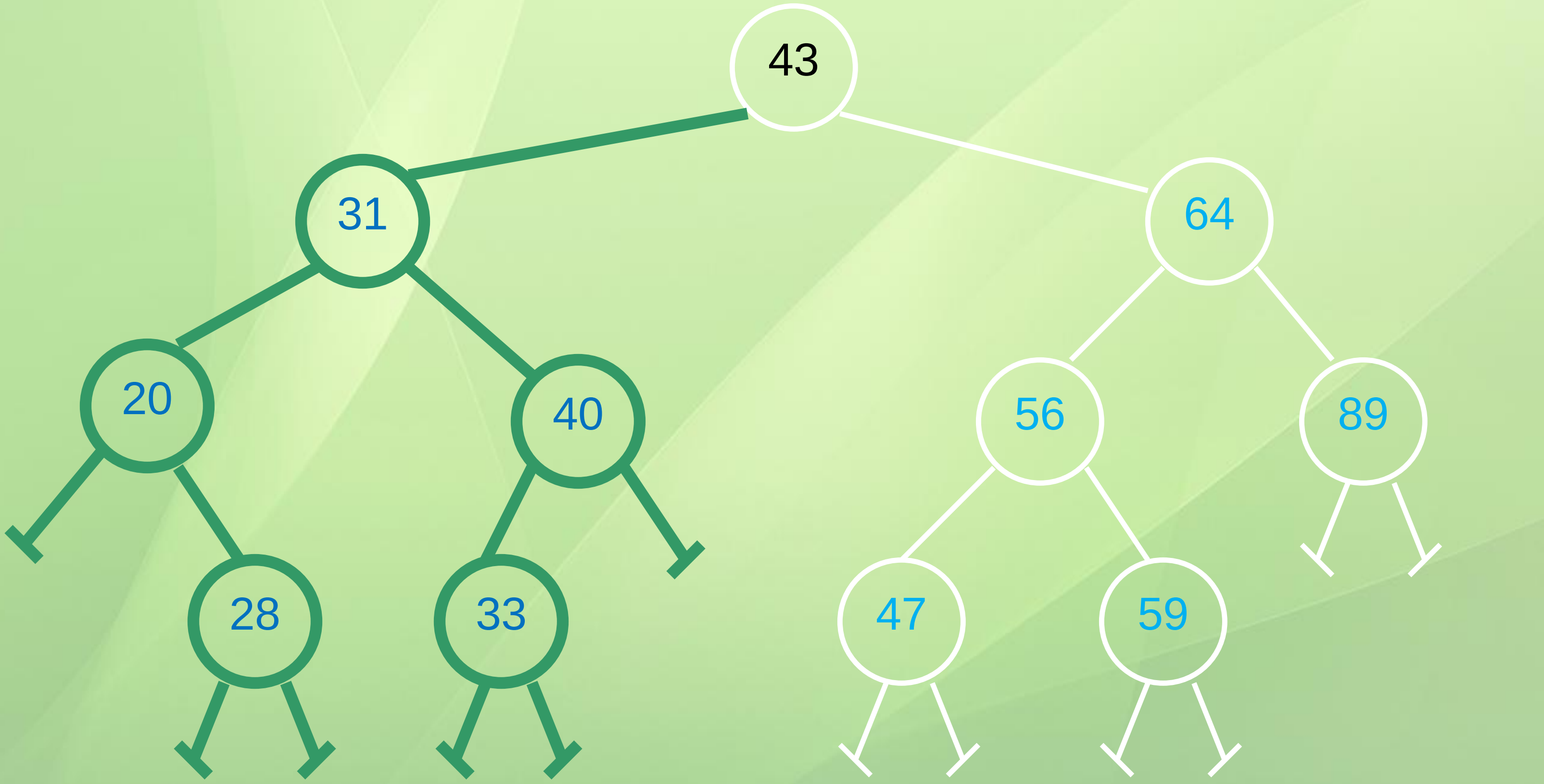
A BST is a Binary Tree such that:

- ❑Every node entry has a unique key or value.
- ❑All the values in the left subtree of a node are smaller(less) than the value of the node.
- ❑All the values in the right subtree of a node are greater than the value of the node.
- ❑Left and rigt sub-trees are also BST.

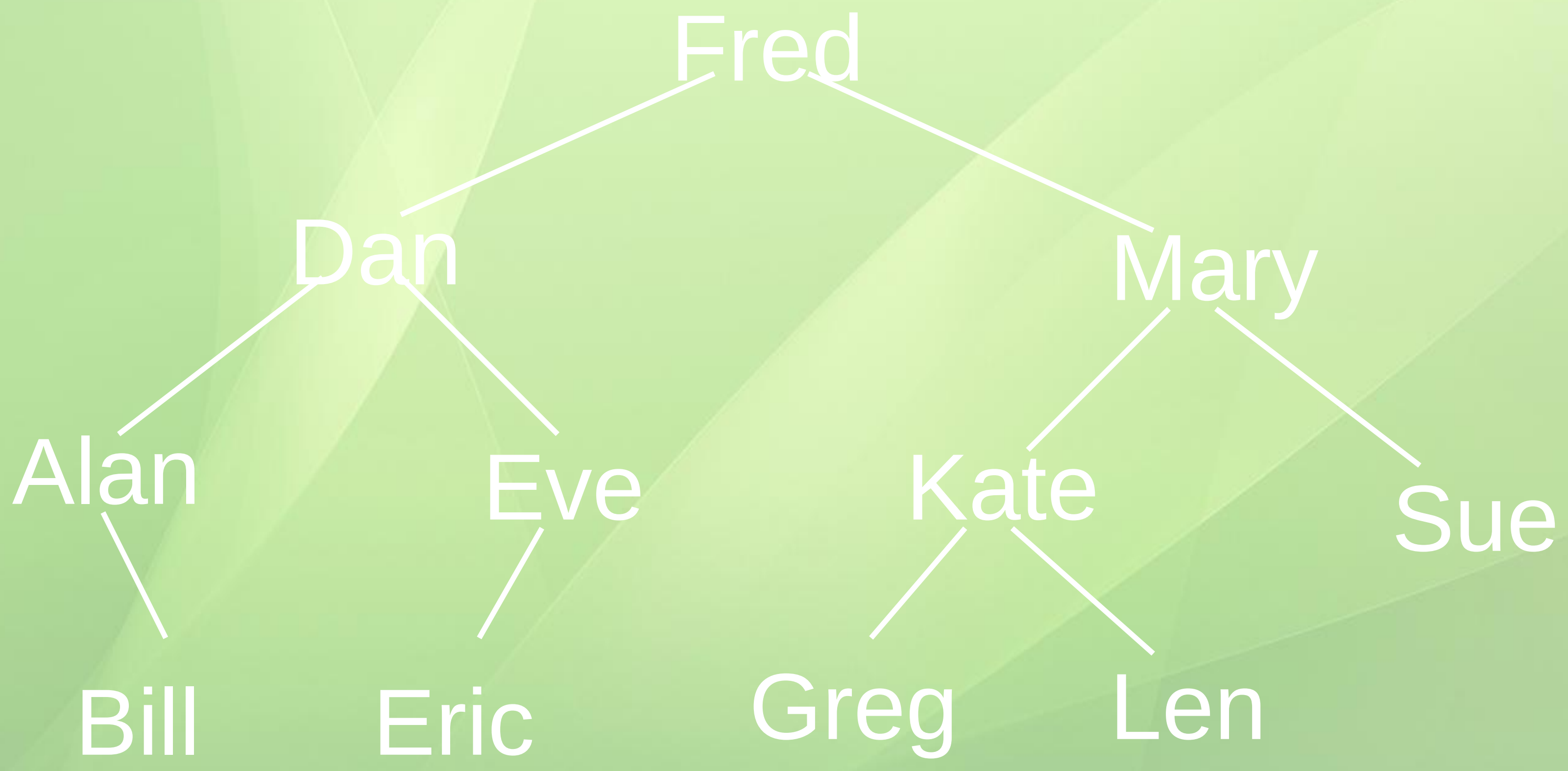
Example 1: *Key or value is an integer*



Example 1: *key or value is an integer*



Example 2: *Key or value is a string*



Creation of BST Using Recursive Function

```
struct node {
 int data;
   node *lchild;
   node *rchild;
};
node *nptr;
 //Recursive Function for BST creation
 node *creatBST(node *root, int item)
    {
      if (root==NULL)
      {
            nptr=new(node);
            nptr->data=item; nptr->lchild=NULL; nptr->rchild=NULL;
           root=nptr;
           return root;
      }
```

Creation of BST Using Recursive Function

```
else
  {
      if (item < root->data)
            root->lchild=creatBST(root->lchild,item);
      else if (item> root->data)
            root->rchild=creatBST(root->rchild,item);
      else cout<<"Duplicate not allowed";
      return root;
  }
  }
```

Main function and Recursive function

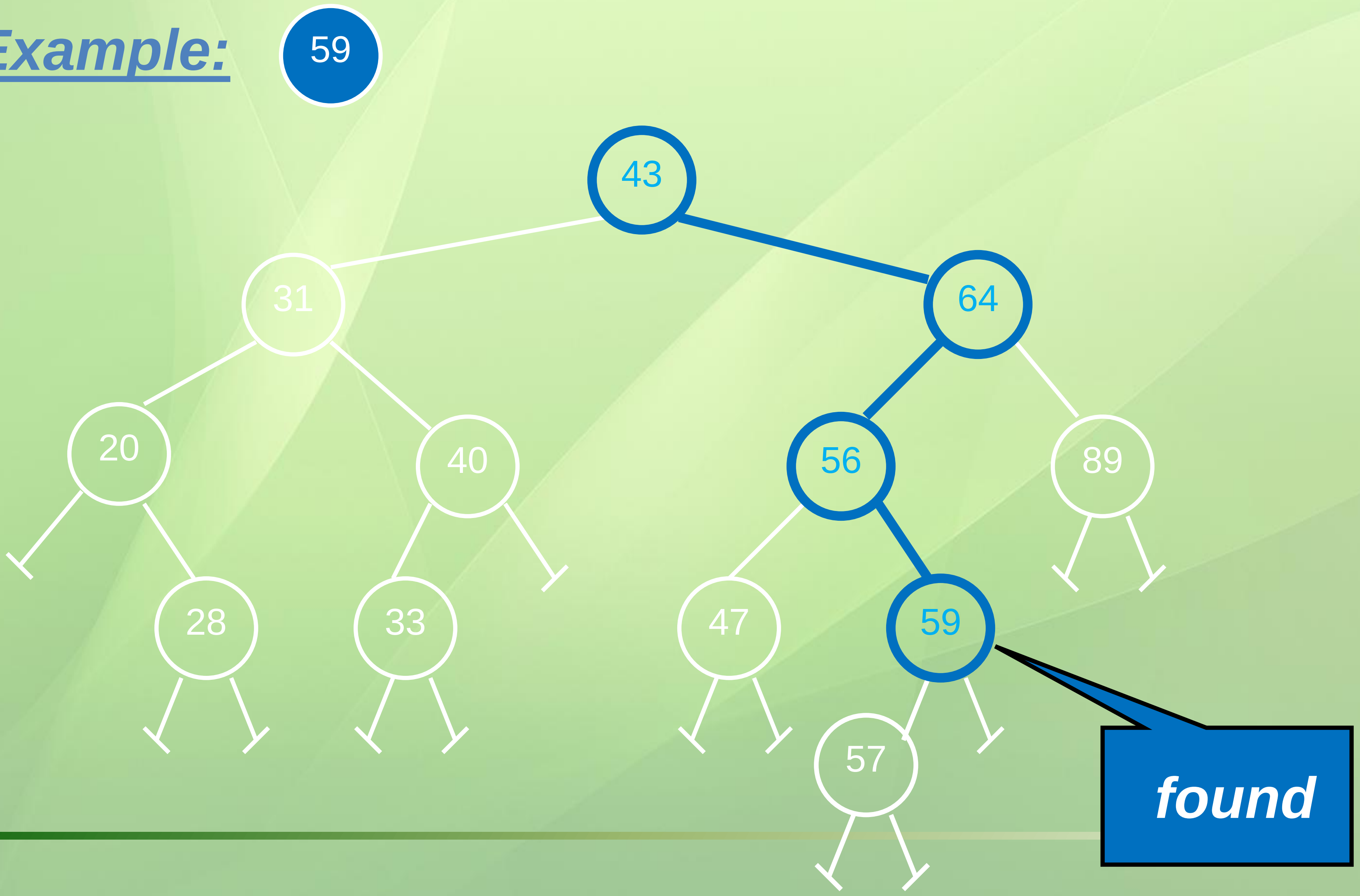
```
int main()
{
    node *root;
    root=NULL;
    int i, item;
    for(i=0; i<5;++i)
    {
        cin>>item;
        root=creatBST(root, item);
    }
    //call inorder( ) to print data
    return 0;
}
```

Searching a particular node

- Compare with the root if equal then found.
- Else if the node value is less search the left subtree.
- Otherwise (for greater) search the right subtree.
- Searching cost is $O(\log_2 n)$ if BST is a complete.

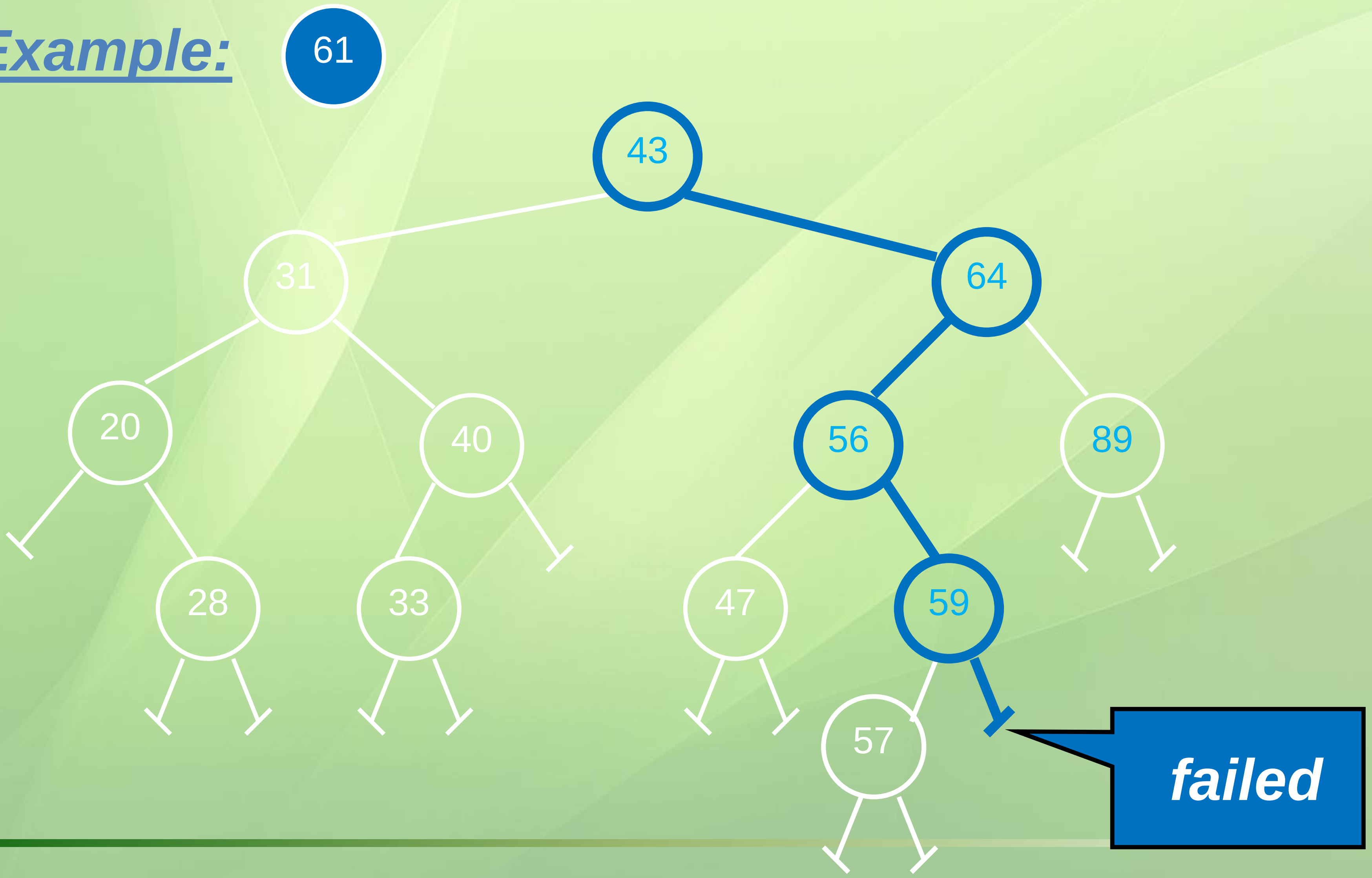
Searching

Example:



Searching

Example:



Searching Algorithm

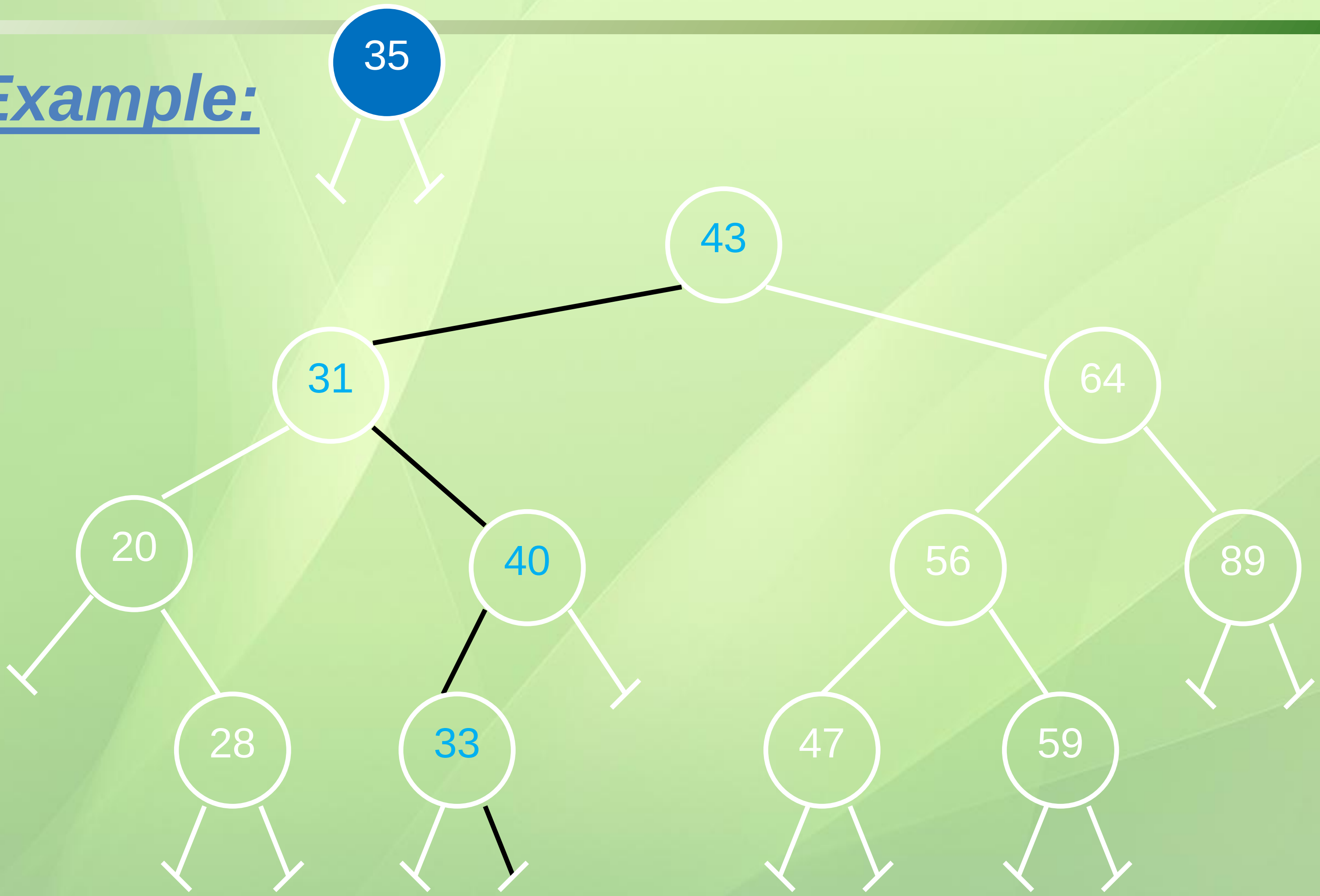
- **Algorithm 7.4:** Algorithm to find out a particular node value of BST
  1. Input BST and a node value, *x*;
  2. Repeat Step-3 to Step-5 until we find the value or we go beyond the tree.
  3. If *x* is equal to root node value, searching is successful (print "Found") and terminate the algorithm.
  4. If *x* is less than root node value, we have to search the left sub-tree (by treating it as a BST).
  5. Else we have to search right sub-tree (by treating it as a BST).
  6. Otherwise the node value is not present in BST (print "Not Found").
  7. Output: Print message "FOUND" or "NOT FOUND"

Insert or Add

- Create new node for the item.
- Find the parent node the node.
- Add or Attach new node as a leaf.

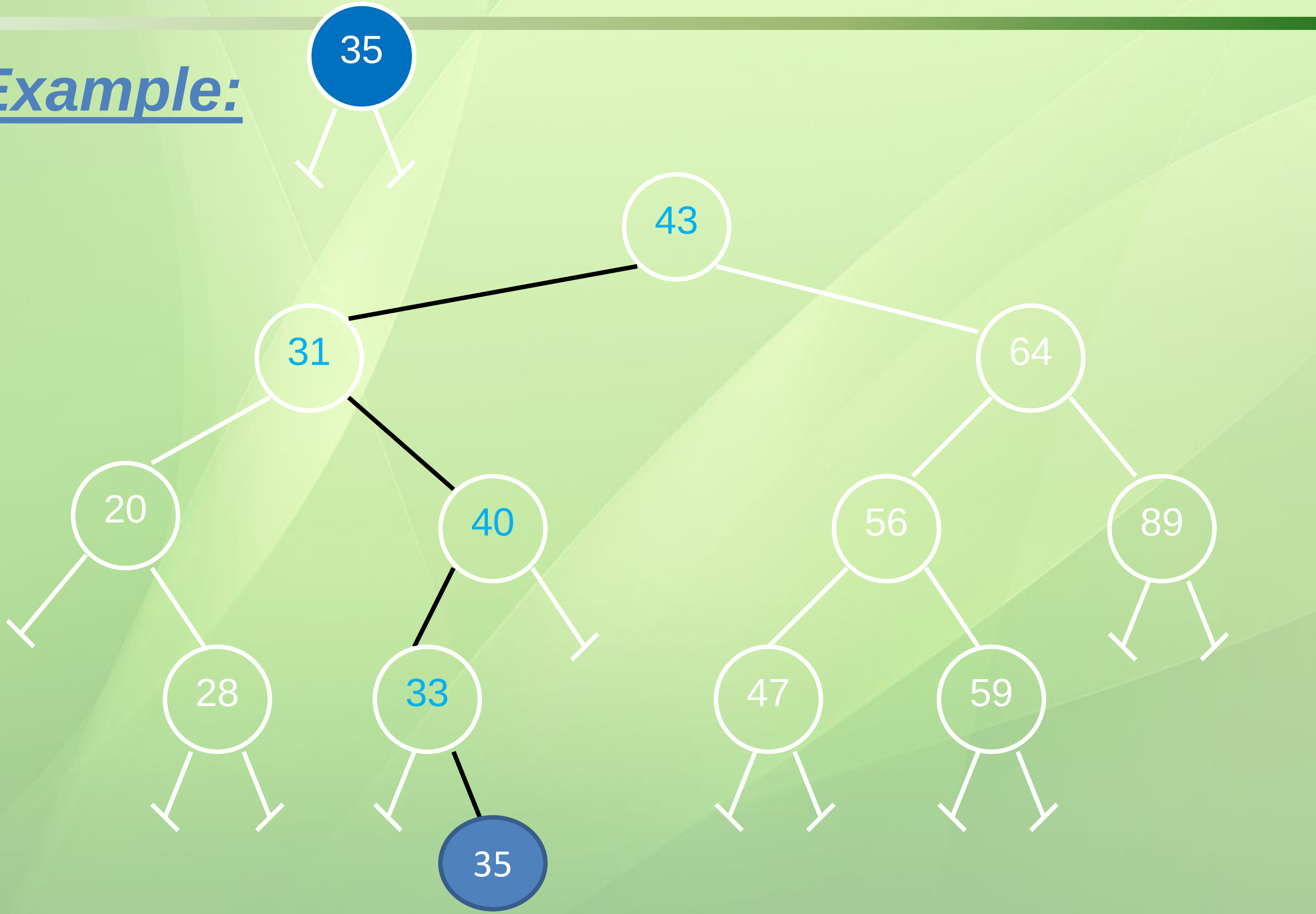
Insert or Add

Example:



Insert or Add

Example:



Algorithm to Add

- **Algorithm 7.5:** Algorithm to insert (add) a node to a BST
    1. Input BST and a new node;
    2. Repeat Step-3 to Step-5 until we find the value or we go beyond the tree.
    3. If x is equal to root node value, searching is successful and terminate the algorithm.
    4. If *x* is less than root node value, we have to search the left sub-tree (by treating it as a BST).
    5. Else we have to search right sub-tree (by treating it as a BST).
    6. If the new node is less the parent node link new node as a left child.
    7. Otherwise link the new node as a right child.
    8. Output: Updated BST.

Delete a node from BST

- Find or locate the node (searching)

Case 1:

- if the target node is a leaf node, then we just delete the node.

Case 2:

- if the target node has only one child, then we make link between the child and parent node of the target node and delete the node.

Deletion from BST

Case 3:

if the target node has two children (with grand children also).

Search left sub-tree of the target node and find the node with maximum value and mark it.

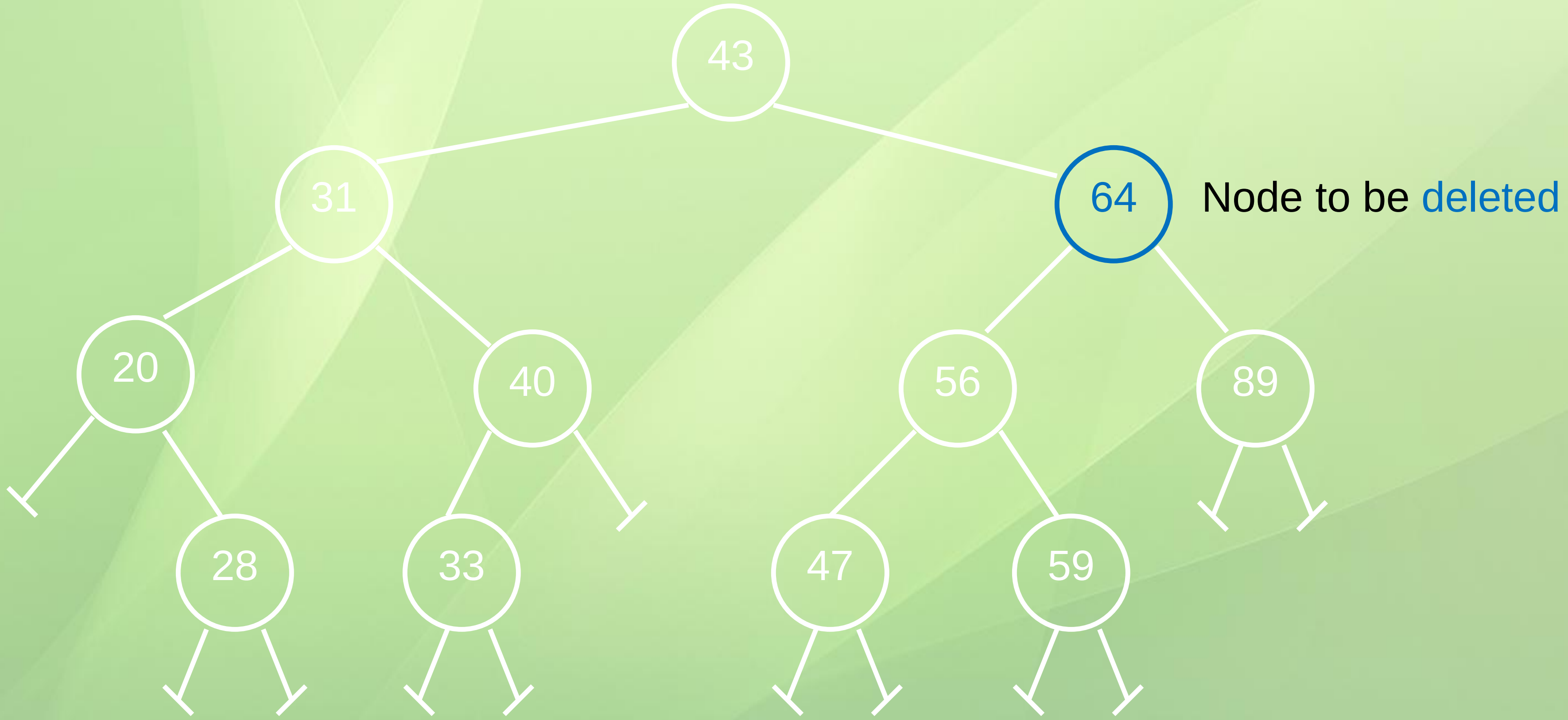
Replace the node value by the maximum value.

Delete the marked node (the node with maximum value).

(We can do it using right subtree also in thet case minimum value will be used).

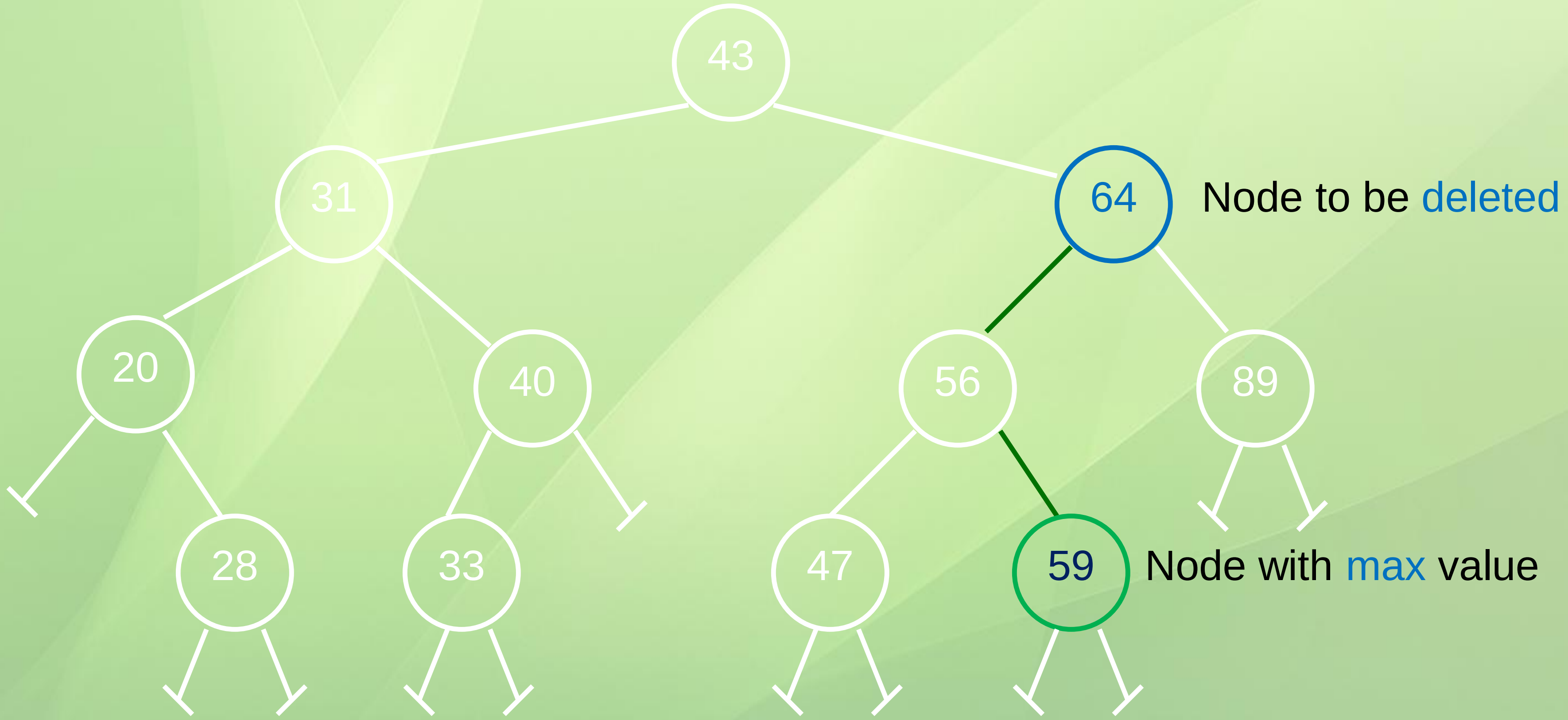
Deletion from BST

Example of Case 3:



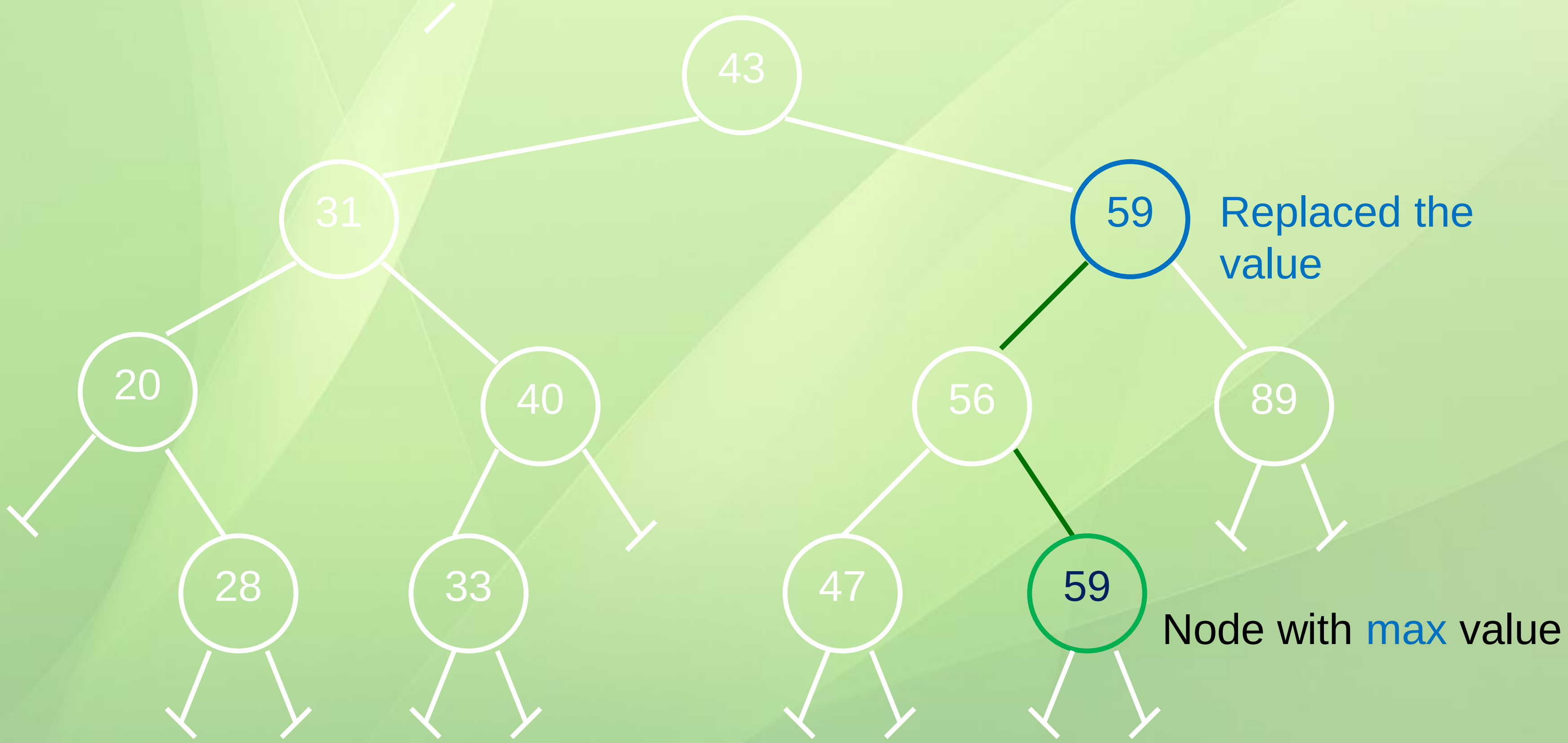
Deletion from BST

Example of case 3:



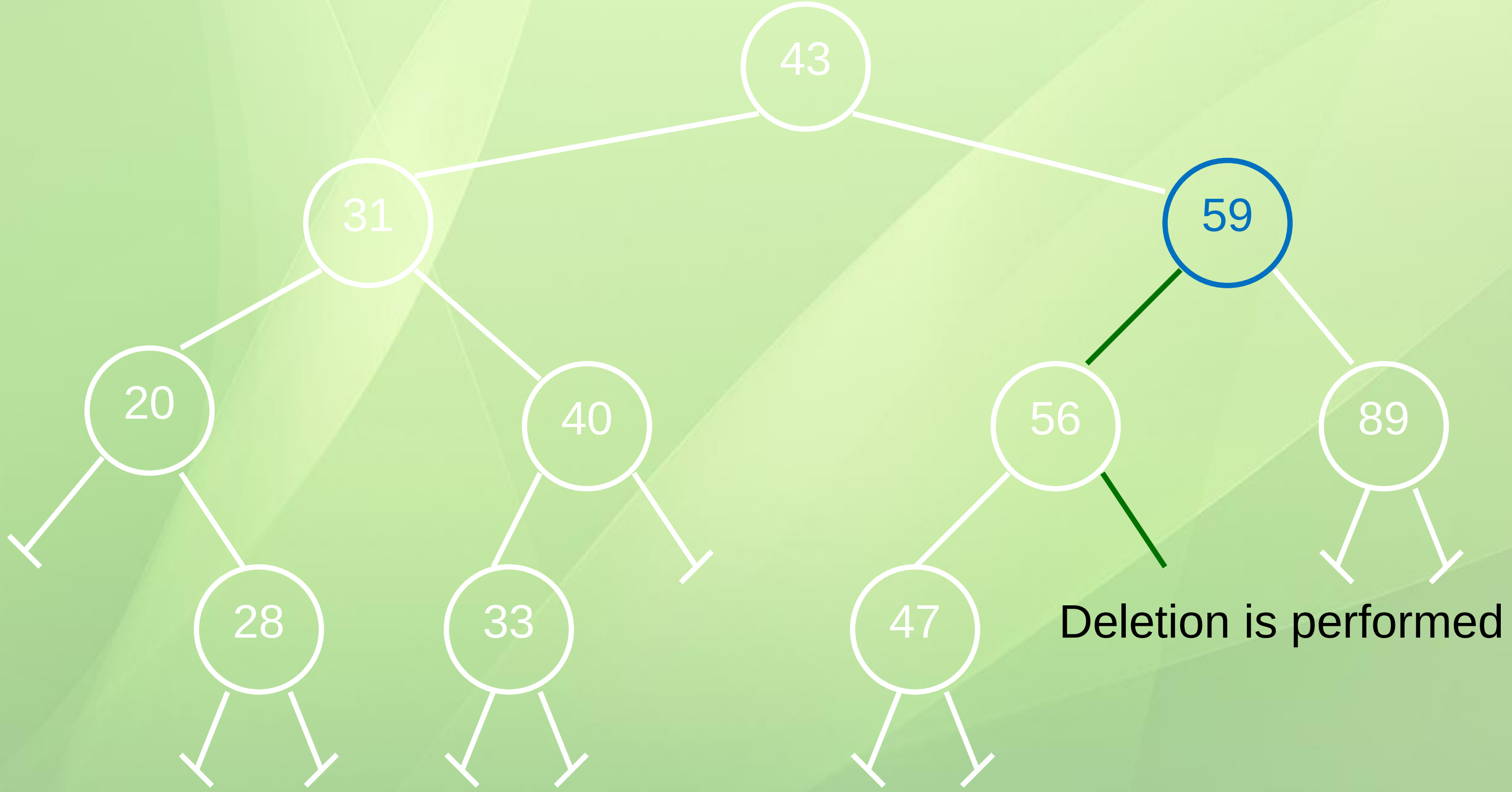
Deletion from BST

Example of case 3:



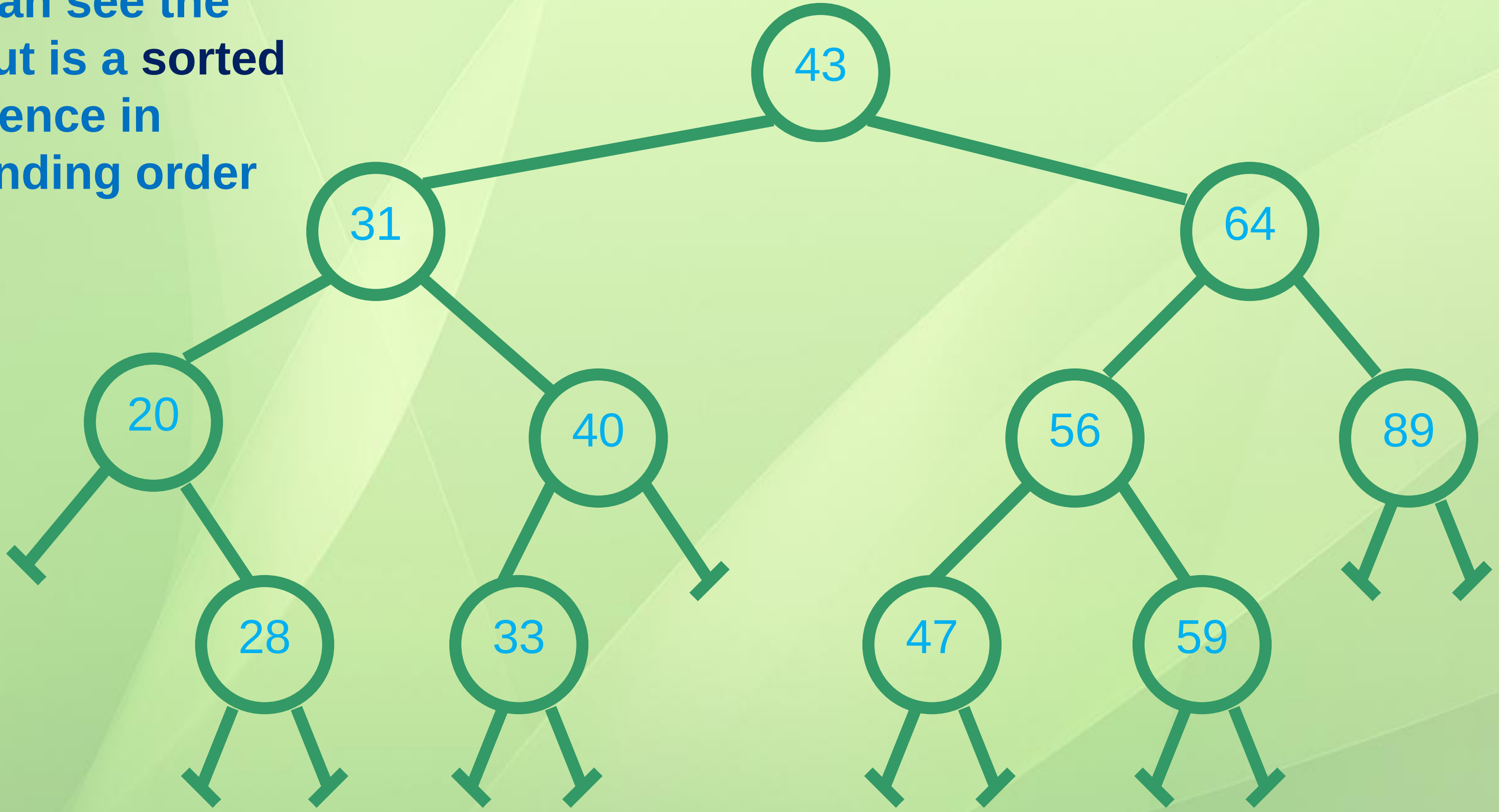
Deletion from BST

Example of Case 3:



Inorder Traversal of a BST

We can see the output is a **sorted** sequence in ascending order

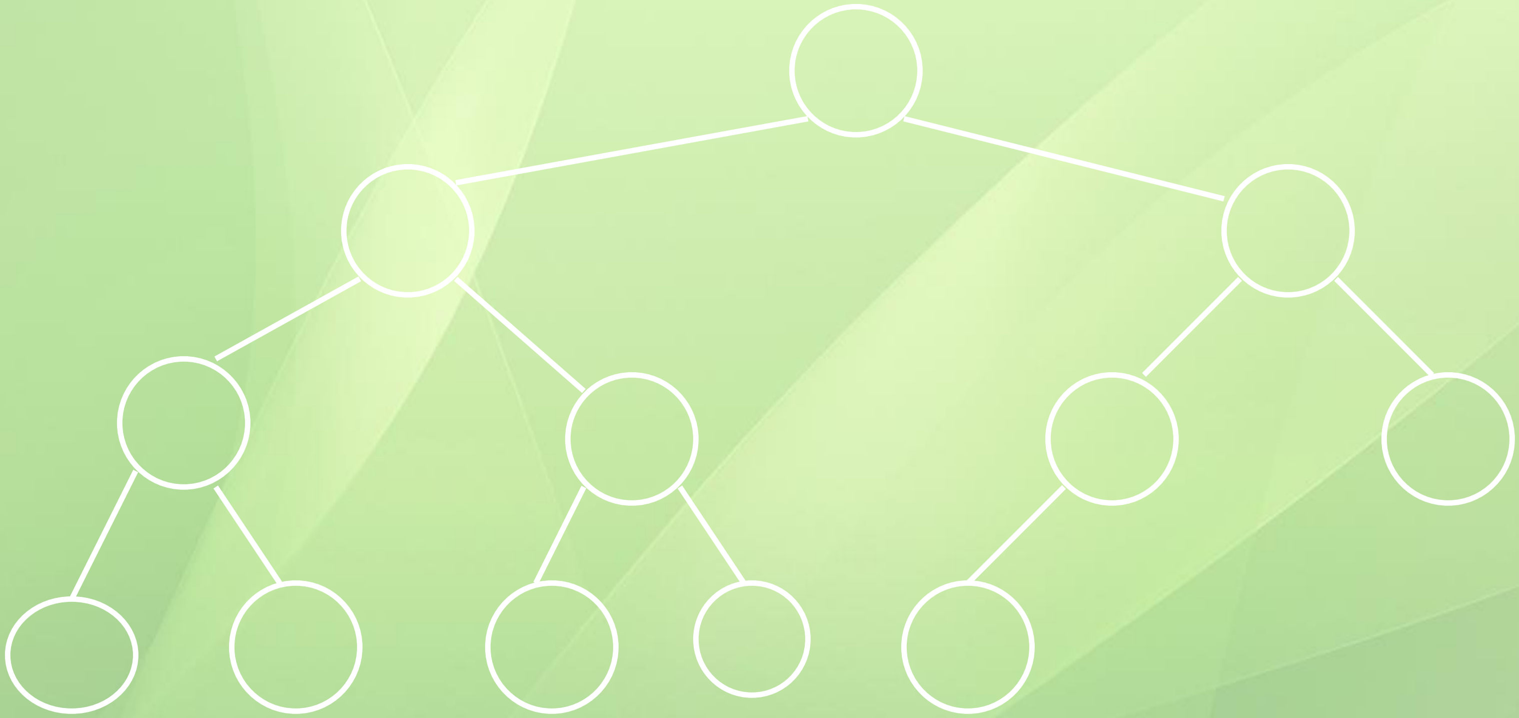


20	28	31	33	40	43	47	56	59	64	89

Heap

- It is a complete binary tree.
- each node value is greater or smaller than the node value of its children.

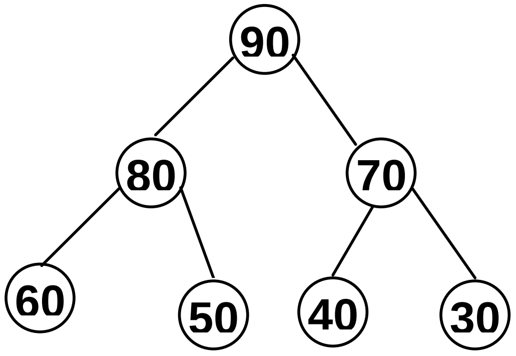
Shape of a Complete Binary Tree



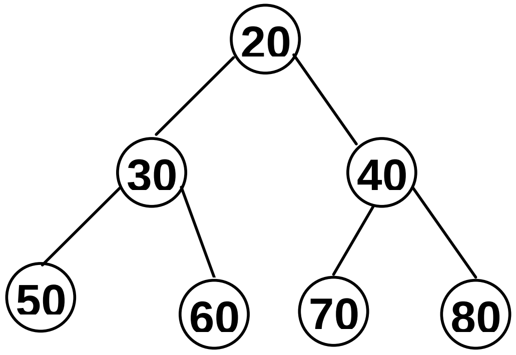
Types of Heap

- ✓ Max-heap and Min-heap
- If the node value is greater than the node value of its children, then the heap is called max heap.
- Otherwise, the heap is called min heap.
- ✓ Array implementation of a heap is efficient.

Heap [contd.]



(a): A max heap



(b): A min heap

Figure 7.12: Graphical representation of Heap

Heap Creation Process

1. Add an element as root of the heap.
2. Select second element as left child and place it in proper position by comparing it with its parent.
3. Select next element as right child and place it in proper position by comparing it with its parent.
4. Select next element as left or right child and place it in proper position by comparing it with is parent and grant parents (if any).

Heap Creation [contd.]

- We have to compare parent and child and check whether parent is smaller if smaller, swap the elements.
- By repeating the process we can have a max-heap.

Heap Creation [contd.]

- Given List: 45 28 52 25 60 70
- Create heap using the above data.

Heap Creation [contd.]

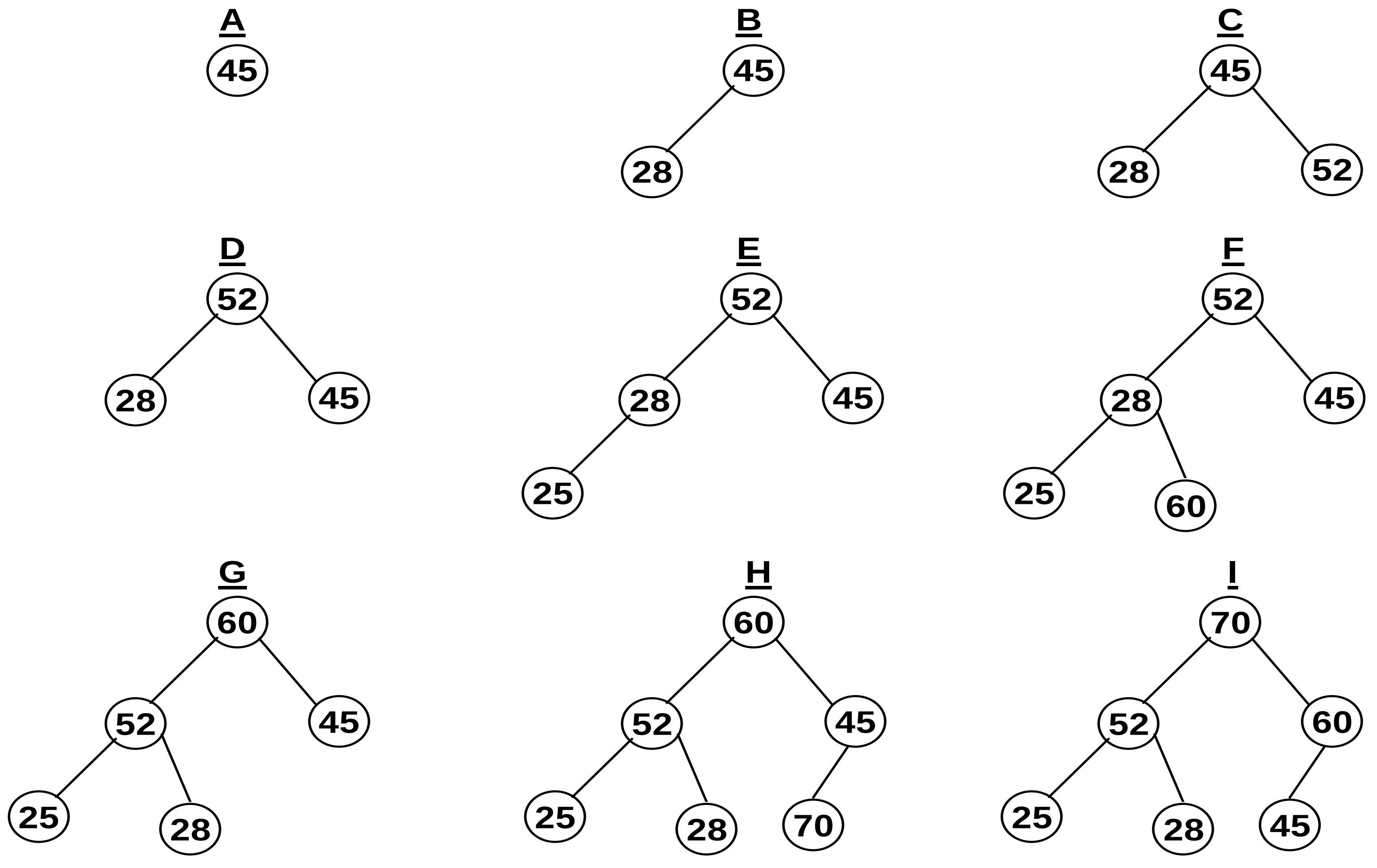


Figure 7.13: Heap creation process (pictorial view)

Algorithm to create a heap (pseudocode)

1. Take an array, *A*[1:*n*] with data and a variable, *temp*

[we shall consider and rearrange data using temp one by one]

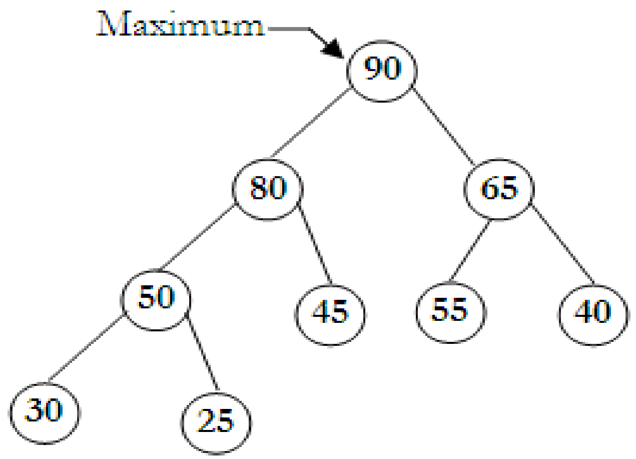
```
    2.  for (i = 2; i<=n; ++i)
    {
        temp=a[i];
    k = i;
    while (k > 0 and A[k/2] < temp)
    {
    A[k] = A[k/2]; //copy parent's value to child's position
    k = k/2;          //consider data of upper/parent's position
    }
    A[k] = temp;
    } // end of for
```

3. Output: a heap

Delete maximum from a max-heap

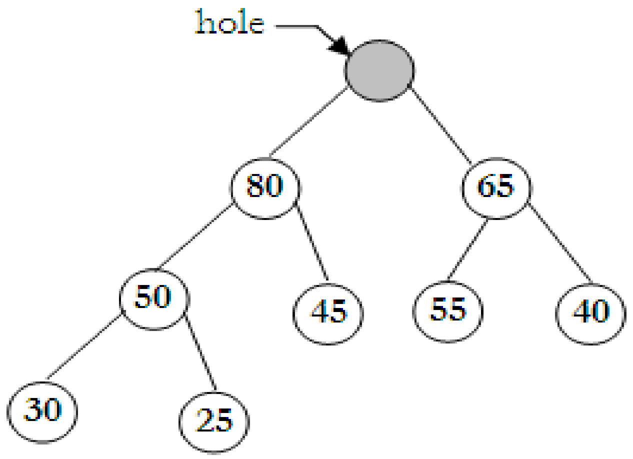
1. Take last data in temporary variable (temp = a[n])
2. Assume there is a hole at the root node.
3. Compare greater child of the hole with data in temp and place the greater one in the hole (root).
4. Repeat step 3 until there is no hole or the hole becomes a leaf node.
5. Place the value of temp in the hole at the last level.

Delete maximum from a max-heap [contd.]

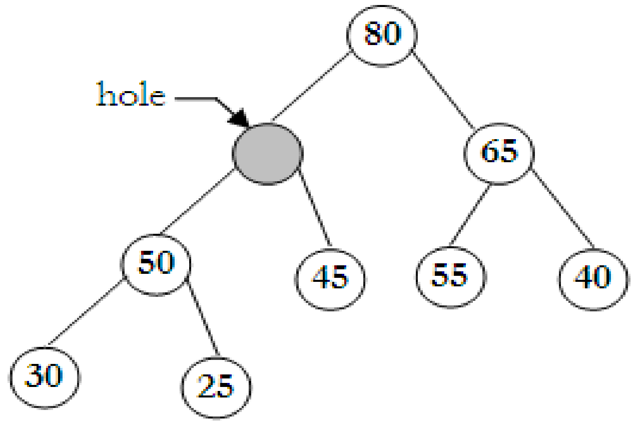


(a) A Max heap

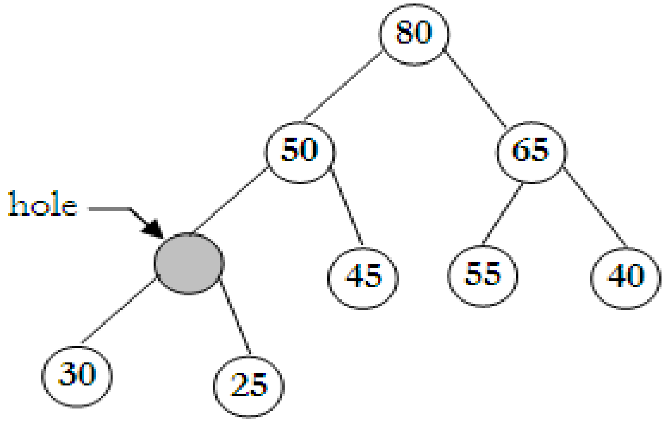
Temp = 25



(b)



(c)



(d)

Delete maximum from a max-heap [contd.]

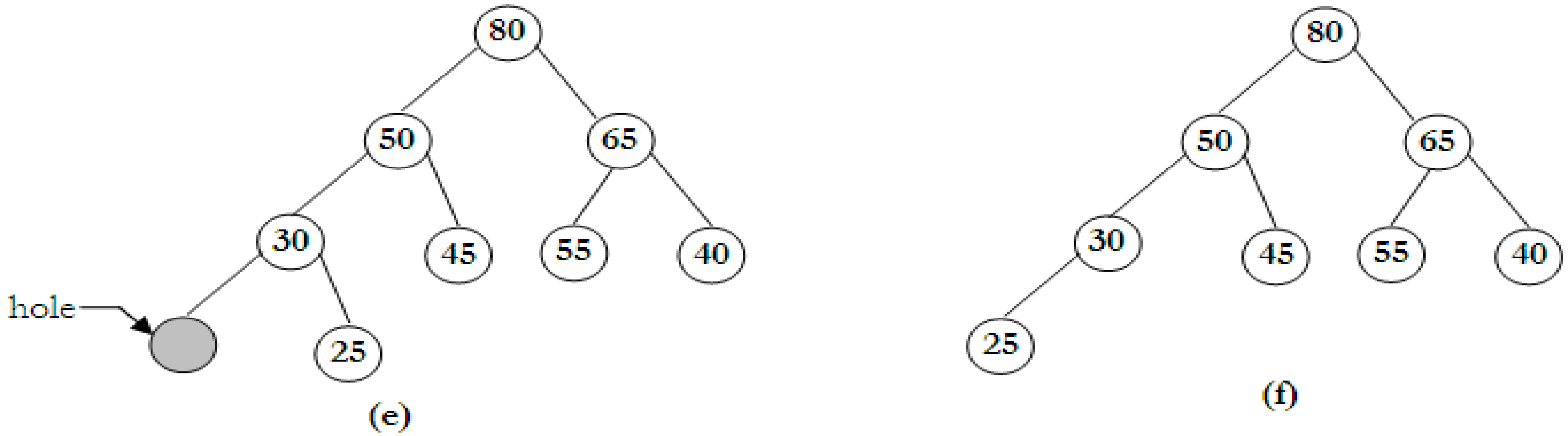


Figure 7.14: Delete the Maximum from a Max heap (pictorial view)

Temp = 25

Put value of temp in hole at last level

Algorithm to delete the Maximum (pseudocode)

```
1.  Input a heap, a[1:n]. // The heap as an array
2.  temp= a[n]; //data of last position is in temp
3.  i = 2;
4.  while (i<=n)
    {
      if ((i<n) and (a[i]<a[i+1])) // find greater child
      i= i+1;
     If (temp>=a[i] break; //if temp is greater than greater child,
             stop
     a[i/2]=a[i]; i=2*I       //copy child's value to parent's position
     }
     a[i/2]=temp;

5. Output updated heap with (n – 1) elements.
```

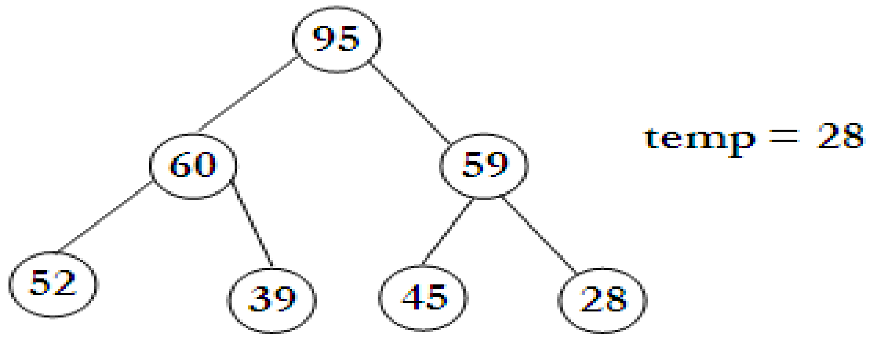
Heapsort

1. Create heap
2. Place the last node value in a temporary place
3. Place the root node value in the last place
4. Rearrange the tree except last node
5. Repeat step 2 to 4.(online)

Heap sort in detail

1. Create a heap with *n* numbers. We can create heap using the heap creation algorithm.
2. Place the value of nth (last) node in a temporary variable.
3. Place the value of the first node (root at present) in the nth position.
4. Treat there is a hole at the root.
5. Compare the children of the hole and place the greater child in the hole.
6. Repeat step-5 until the hole is a leaf node.
7. Place the value in the temporary place in the (last) hole.
8. Consider the rest of the data (which are in a heap) except last one.
9. Repeat the step-2 to step-7 until there is only one element in the heap.

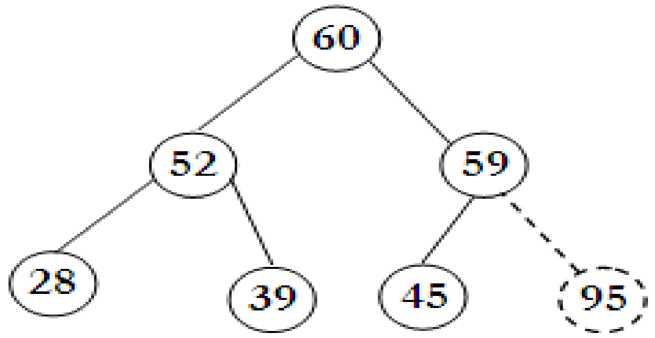
Heap sort [contd.]



A Max heap with n elements

Array:							
	95	60	59	52	39	45	28

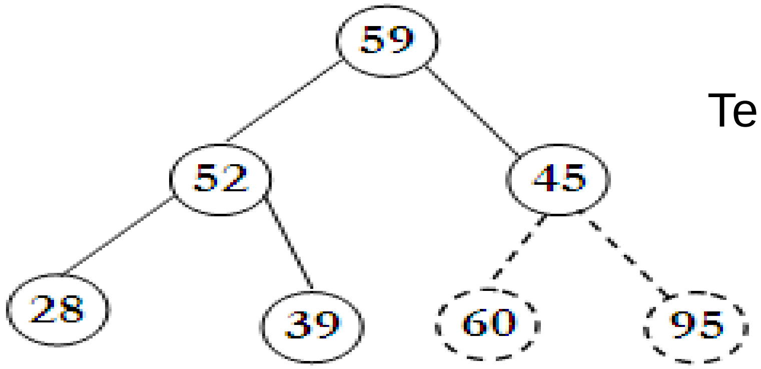
a) A heap and its corresponding array



Array:							
	60	52	59	28	39	45	95

b) First step of sorting process using heap

Heap sort [contd.]



Temp =45

Array:

59	52	45	28	39	60	95

c) Second step of sorting process using heap

--- --- --- --- --- ---
--- --- --- --- --- ---
--- --- --- --- --- ---

Array:

28	39	45	52	59	60	95

d) Sorted data

Figure 7.15: Pictorial view of heap sorting process

Heap Sort

1. Input an array, a[1:n] with random data
2. heapify (a, n);
3. for (k = n; k>1; --k)

   rearange (a, k);
4. Output a sorted list in a [1:n];

Here heapify and rearange are two functions. Then function heapify builds a heap and rearange function adjusts data for sorting.

THANK YOU