# Binarytree Module in Python

In [computer science](https://en.wikipedia.org/wiki/Computer_science), a binary tree is a [tree data structure](https://en.wikipedia.org/wiki/Tree_(data_structure)) in which each node has at most two [children](https://en.wikipedia.org/wiki/Child_node), which are referred to as the left child and the right child. A [recursive definition](https://en.wikipedia.org/wiki/Recursive_definition) using just [set theory](https://en.wikipedia.org/wiki/Set_theory) notions is that a (non-empty) binary tree is a [tuple](https://en.wikipedia.org/wiki/Tuple) (L, S, R), where L and R are binary trees or the [empty set](https://en.wikipedia.org/wiki/Empty_set) and S is a [singleton set](https://en.wikipedia.org/wiki/Singleton_set) containing the root.[]](https://en.wikipedia.org/wiki/Binary_tree#cite_note-GarnierTaylor2009-1) Some authors allow the binary tree to be the empty set as well.

From a [graph theory](https://en.wikipedia.org/wiki/Graph_theory) perspective, binary (and K-ary) trees as defined here are [arborescences](https://en.wikipedia.org/wiki/Arborescence_(graph_theory)). A binary tree may thus be also called a bifurcating arborescence—a term which appears in some very old programming books, before the modern computer science terminology prevailed. It is also possible to interpret a binary tree as an [undirected](https://en.wikipedia.org/wiki/Undirected_graph), rather than a [directed graph](https://en.wikipedia.org/wiki/Directed_graph), in which case a binary tree is an [ordered](https://en.wikipedia.org/wiki/Ordered_tree), [rooted tree](https://en.wikipedia.org/wiki/Rooted_tree). Some authors use rooted binary tree instead of binary tree to emphasize the fact that the tree is rooted, but as defined above, a binary tree is always rooted. A binary tree is a special case of an ordered [K-ary tree](https://en.wikipedia.org/wiki/K-ary_tree), where K is 2.

In mathematics, what is termed binary tree can vary significantly from author to author. Some use the definition commonly used in computer science, but others define it as every non-leaf having exactly two children and don't necessarily order (as left/right) the children either.

In computing, binary trees are used in two very different ways:

* First, as a means of accessing nodes based on some value or label associated with each node. Binary trees labelled this way are used to implement [binary search trees](https://en.wikipedia.org/wiki/Binary_search_tree) and [binary heaps](https://en.wikipedia.org/wiki/Binary_heap), and are used for efficient [searching](https://en.wikipedia.org/wiki/Search_algorithm) and [sorting](https://en.wikipedia.org/wiki/Sorting_algorithm). The designation of non-root nodes as left or right child even when there is only one child present matters in some of these applications, in particular, it is significant in binary search trees. However, the arrangement of particular nodes into the tree is not part of the conceptual information. For example, in a normal binary search tree the placement of nodes depends almost entirely on the order in which they were added, and can be re-arranged (for example by [balancing](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree)) without changing the meaning.
* Second, as a representation of data with a relevant bifurcating structure. In such cases, the particular arrangement of nodes under and/or to the left or right of other nodes is part of the information (that is, changing it would change the meaning). Common examples occur with [Huffman coding](https://en.wikipedia.org/wiki/Huffman_coding) and [cladograms](https://en.wikipedia.org/wiki/Cladograms). The everyday division of documents into chapters, sections, paragraphs, and so on is an analogous example with n-ary rather than binary trees.

A binary tree is a data structure in which every node or vertex has atmost two children. In Python, a binary tree can be represented in different ways with different data structures(dictionary, list) and class representation for a node. However, binarytree library helps to directly implement a binary tree. It also supports heap and binary search tree(BST). This module does not come pre-installed with Python’s standard utility module. To install it type the below command in the terminal.

The node class represents the structure of a particular node in the binary tree. The attributes of this class are values, left, right.

**Syntax:** binarytree.Node(value, left=None, right=None)  
**Parameters:**   
**value:** Contains the data for a node. This value must be number.   
**left:** Conatins the details of left node child.   
**right:** Contains details of the right node child. 

**Example:**

* Python3

|  |
| --- |
| from binarytree import Node  root = Node(3)  root.left = Node(6)  root.right = Node(8)    # Getting binary tree  print('Binary tree :', root)    # Getting list of nodes  print('List of nodes :', list(root))    # Getting inorder of nodes  print('Inorder of nodes :', root.inorder)    # Checking tree properties  print('Size of tree :', root.size)  print('Height of tree :', root.height)    # Get all properties at once  print('Properties of tree : \n', root.properties) |

**Output:**

Binary tree :   
3   
/ \   
6 8  
List of nodes : [Node(3), Node(6), Node(8)]  
Inorder of nodes : [Node(6), Node(3), Node(8)]  
Size of tree : 3  
Height of tree : 1  
Properties of tree :   
{‘height’: 1, ‘size’: 3, ‘is\_max\_heap’: False, ‘is\_min\_heap’: True, ‘is\_perfect’: True, ‘is\_strict’: True, ‘is\_complete’: True, ‘leaf\_count’: 2, ‘min\_node\_value’: 3, ‘max\_node\_value’: 8, ‘min\_leaf\_depth’: 1, ‘max\_leaf\_depth’: 1, ‘is\_bst’: False, ‘is\_balanced’: True, ‘is\_symmetric’: False}

#### Build a binary tree from the List:

Instead of using the Node method repeatedly, we can use build() method to convert a list of values into a binary tree.   
Here, a given list contains the nodes of tree such that the element at index i has its left child at index 2\*i+1, the right child at index 2\*i+2 and parent at (i – 1)//2. The elements at index j for j>len(list)//2 are leaf nodes. None indicates the absence of a node at that index. We can also get the list of nodes back after building a binary tree using **values** attribute.

**Syntax:** binarytree.build(values)  
**Parameters:**   
**values:** List representation of the binary tree.  
**Returns:** root of the binary tree. 

**Example:**

* Python3

|  |
| --- |
| # Creating binary tree  # from given list  from binarytree import build      # List of nodes  nodes =[3, 6, 8, 2, 11, None, 13]    # Building the binary tree  binary\_tree = build(nodes)  print('Binary tree from list :\n',        binary\_tree)    # Getting list of nodes from  # binarytree  print('\nList from binary tree :',        binary\_tree.values) |

**Output:**

Binary tree from list :

\_\_\_3

/ \

6 8

/ \ \

2 11 13

List from binary tree : [3, 6, 8, 2, 11, None, 13]

#### Build a random binary tree:

tree() generates a random binary tree and returns its root node.

**Syntax:** binarytree.tree(height=3, is\_perfect=False)  
**Parameters:**   
**height:** It is the height of the tree and its value can be between the range 0-9 (inclusive)   
**is\_perfect:** If set True a perfect binary is created.  
**Returns:** Root node of the binary tree. 

**Example:**

* Python3

|  |
| --- |
| from binarytree import tree      # Create a random binary  # tree of any height  root = tree()  print("Binary tree of any height :")  print(root)    # Create a random binary  # tree of given height  root2 = tree(height = 2)  print("Binary tree of given height :")  print(root2)    # Create a random perfect  # binary tree of given height  root3 = tree(height = 2,               is\_perfect = True)  print("Perfect binary tree of given height :")  print(root3) |

**Output:**

Binary tree of any height :

14\_\_\_\_

/ \

2 5\_\_

/ / \

6 1 13

/ / / \

7 9 4 8

Binary tree of given height :

1\_\_

/ \

5 2

/ \

4 3

Perfect binary tree of given height :

\_\_3\_\_

/ \

2 4

/ \ / \

6 0 1 5

#### Building a BST:

The binary search tree is a special type of tree data structure whose inorder gives a sorted list of nodes or vertices. In Python, we can directly create a BST object using binarytree module. bst() generates a random binary search tree and return its root node.

**Syntax:** binarytree.bst(height=3, is\_perfect=False)  
**Parameters:**   
**height:** It is the height of the tree and its value can be between the range 0-9 (inclusive)   
**is\_perfect:** If set True a perfect binary is created.  
**Returns:** Root node of the BST. 

**Example:**

* Python3

|  |
| --- |
| from binarytree import bst      # Create a random BST  # of any height  root = bst()  print('BST of any height : \n',        root)    # Create a random BST of  # given height  root2 = bst(height = 2)  print('BST of given height : \n',        root2)    # Create a random perfect  # BST of given height  root3 = bst(height = 2,              is\_perfect = True)  print('Perfect BST of given height : \n',        root3) |

**Output:**

BST of any height :

\_\_\_\_9\_\_\_\_\_\_

/ \

\_\_5\_\_ \_\_\_\_12\_\_\_

/ \ / \

2 8 10 \_14

/ \ / \ /

1 4 7 11 13

BST of given height :

5

/ \

4 6

/

3

Perfect BST of given height :

\_\_3\_\_

/ \

1 5

/ \ / \

0 2 4 6

#### Importing heap:

Heap is a tree data structure that can be of two types –

* max heap
* min heap

Using the heap() method of binarytree library, we can generate a random maxheap and return its root node. To generate minheap, we need to set the is\_max attribute as False.

**Syntax:** binarytree.heap(height=3, is\_max=True, is\_perfect=False)  
**Parameters:**   
**height:** It is the height of the tree and its value can be between the range 0-9 (inclusive)   
**is\_max:** If set True generates a max heap else min heap.   
**is\_perfect:** If set True a perfect binary is created.  
**Returns:** Root node of the heap. 

* Python3

|  |
| --- |
| from binarytree import heap      # Create a random max-heap  root = heap()  print('Max-heap of any height : \n',        root)    # Create a random max-heap  # of given height  root2 = heap(height = 2)    print('Max-heap of given height : \n',        root2)    # Create a random perfect  # min-heap of given height  root3 = heap(height = 2,               is\_max = False,               is\_perfect = True)    print('Perfect min-heap of given height : \n',        root3) |

**Output:**

Max-heap of any height :

\_\_\_\_\_\_\_14\_\_\_\_\_\_

/ \

\_\_\_12\_\_ \_\_13\_\_

/ \ / \

10 8 3 9

/ \ / \ / \ /

1 5 4 6 0 2 7

Max-heap of given height :

\_\_6\_\_

/ \

4 5

/ \ / \

2 0 1 3

Perfect min-heap of given height :

\_\_0\_\_

/ \

1 3

/ \ / \

2 6 4 5