binarytree

Jun 28, 2022

Contents

Welcome to the documentation for **binarytree**.

Binarytree is Python library which lets you generate, visualize, inspect and manipulate binary trees. Skip the tedious work of setting up test data, and dive straight into practising algorithms. Heaps and binary search trees are also supported. Self-balancing search trees like red-black or AVL will be added in the future.

CHAPTER 1

Requirements

Python 3.7+

CHAPTER 2

Installation

Install via pip:

```
pip install binarytree --upgrade
```

For conda users:

```
conda install binarytree -c conda-forge
```

CHAPTER 3

Contents

3.1 Overview

Binarytree uses the following class to represent a node:

```
class Node:

    def __init__(self, value, left=None, right=None):
        self.value = value  # The node value (float/int/str)
        self.left = left    # Left child
        self.right = right  # Right child
```

Generate and pretty-print various types of binary trees:

```
>>> from binarytree import tree, bst, heap
>>>
>>> # Generate a random binary tree and return its root node.
>>> my_tree = tree(height=3, is_perfect=False)
>>>
>>> # Generate a random BST and return its root node.
>>> my_bst = bst(height=3, is_perfect=True)
>>>
>>> # Generate a random max heap and return its root node.
>>> my_heap = heap(height=3, is_max=True, is_perfect=False)
>>>
>>> # Pretty-print the trees in stdout.
>>> print(my_tree)

    ________1_____
   /              \
  4__           ___3
 /   \         /    \
0     9       13     14
     / \       \
```

(continues on next page)

(continued from previous page)

```
    7   10      2

>>> print(my_bst)

        _______7________
       /                \
    __3__             ___11___
   /     \           /         \
  1       5         9          _13
 / \     / \       / \        /   \
0   2   4   6     8   10     12    14

>>> print(my_heap)

          ______14__
         /           \
    _____13__         9
   /         \       / \
  12          7     3   8
 /  \        /
0    10     6
```

Generate trees with letter values instead of numbers:

```
>>> from binarytree import tree

>>> my_tree = tree(height=3, is_perfect=False, letters=True)

>>> print(my_tree)

      _____H_____
     /           \
  __E__           F__
 /     \         /   \
M       G       J     B
 \     /       /     / \
  O   L       D     I   A
```

Build your own trees:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> print(root)

  __1
 /   \
2     3
 \
  4
```

Inspect tree properties:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.height
2
>>> root.is_balanced
True
>>> root.is_bst
False
>>> root.is_complete
True
>>> root.is_max_heap
False
>>> root.is_min_heap
True
>>> root.is_perfect
False
>>> root.is_strict
True
>>> root.leaf_count
3
>>> root.max_leaf_depth
2
>>> root.max_node_value
5
>>> root.min_leaf_depth
1
>>> root.min_node_value
1
>>> root.size
5

>>> properties = root.properties  # Get all properties at once.
>>> properties['height']
2
>>> properties['is_balanced']
True
>>> properties['max_leaf_depth']
2

>>> root.leaves
[Node(3), Node(4), Node(5)]

>>> root.levels
[[Node(1)], [Node(2), Node(3)], [Node(4), Node(5)]]
```

Compare and clone trees:

```
>>> from binarytree import tree
>>> original = tree()
>>> clone = original.clone()
>>> original.equals(clone)
True
```

Use level-order (breadth-first) indexes to manipulate nodes:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)                    # index: 0, value: 1
>>> root.left = Node(2)               # index: 1, value: 2
>>> root.right = Node(3)              # index: 2, value: 3
>>> root.left.right = Node(4)         # index: 4, value: 4
>>> root.left.right.left = Node(5)    # index: 9, value: 5
>>>
>>> print(root)

  _____1
 /      \
2__      3
   \
    4
   /
  5

>>> # Use binarytree.Node.pprint instead of print to display indexes.
>>> root.pprint(index=True)

   ___________0-1_
  /               \
1-2_____          2-3
        \
       _4-4
      /
    9-5

>>> # Return the node/subtree at index 9.
>>> root[9]
Node(5)

>>> # Replace the node/subtree at index 4.
>>> root[4] = Node(6, left=Node(7), right=Node(8))
>>> root.pprint(index=True)

   _________________0-1_
  /                     \
1-2_____                2-3
        \
       _4-6_
      /     \
    9-7     10-8

>>> # Delete the node/subtree at index 1.
>>> del root[1]
>>> root.pprint(index=True)
```

(continues on next page)

(continued from previous page)

```
0-1_
    \
    2-3
```

Traverse trees using different algorithms:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.inorder
[Node(4), Node(2), Node(5), Node(1), Node(3)]

>>> root.preorder
[Node(1), Node(2), Node(4), Node(5), Node(3)]

>>> root.postorder
[Node(4), Node(5), Node(2), Node(3), Node(1)]

>>> root.levelorder
[Node(1), Node(2), Node(3), Node(4), Node(5)]

>>> list(root)  # Equivalent to root.levelorder.
[Node(1), Node(2), Node(3), Node(4), Node(5)]
```

Convert to List representations:

```
>>> from binarytree import build
>>>
>>> # Build a tree from list representation.
>>> values = [7, 3, 2, 6, 9, None, 1, 5, 8]
>>> root = build(values)
>>> print(root)

        __7
       /   \
    __3     2
   /   \     \
  6     9     1
 / \
5   8

>>> # Convert the tree back to list representation.
>>> root.values
```

(continues on next page)

(continued from previous page)

```
[7, 3, 2, 6, 9, None, 1, 5, 8]
```

Binarytree supports another representation which is more compact but without the indexing properties (this method is often used in Leetcode):

```
>>> from binarytree import build, build2, Node
>>>
>>> # First let's create an example tree.
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.left.left = Node(3)
>>> root.left.left.left = Node(4)
>>> root.left.left.right = Node(5)
>>> print(root)

        1
       /
    __2
   /
  3
 / \
4   5


>>> # First representation is already shown above.
>>> # All "null" nodes in each level are present.
>>> root.values
[1, 2, None, 3, None, None, None, 4, 5]

>>> # Second representation is more compact but without the indexing properties.
>>> root.values2
[1, 2, None, 3, None, 4, 5]

>>> # Build trees from both list representations.
>>> tree1 = build(root.values)
>>> tree2 = build2(root.values2)
>>> tree1.equals(tree2)
True
```

See *API Specification* for more details.

3.2 API Specification

This page covers the API specification for the following classes and utility functions:

- *binarytree.Node*
- *binarytree.build()*
- *binarytree.build2()*
- *binarytree.tree()*
- *binarytree.bst()*
- *binarytree.heap()*
- *binarytree.get_parent()*

3.2.1 Class: binarytree.Node

class binarytree.**Node**(*value: Any*, *left: Optional[Node] = None*, *right: Optional[Node] = None*)

Represents a binary tree node.

This class provides methods and properties for managing the current node, and the binary tree in which the node is the root. When a docstring in this class mentions "binary tree", it is referring to the current node and its descendants.

Parameters

- **value** (*float | int | str*) – Node value (must be a float/int/str).
- **left** (*binarytree.Node | None*) – Left child node (default: None).
- **right** (*binarytree.Node | None*) – Right child node (default: None).

Raises

- ***binarytree.exceptions.NodeTypeError*** – If left or right child node is not an instance of *binarytree.Node*.
- ***binarytree.exceptions.NodeValueError*** – If node value is invalid.

__delitem__(*index: int*) → None

Remove the node (or subtree) at the given level-order index.

- An exception is raised if the target node is missing.
- The descendants of the target node (if any) are also removed.
- Root node (current node) cannot be deleted.

Parameters **index** (*int*) – Level-order index of the node.

Raises

- ***binarytree.exceptions.NodeNotFoundError*** – If the target node or its parent is missing.
- ***binarytree.exceptions.NodeModifyError*** – If user attempts to delete the root node (current node).

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)           # index: 0, value: 1
>>> root.left = Node(2)      # index: 1, value: 2
>>> root.right = Node(3)     # index: 2, value: 3
>>>
>>> del root[0]
Traceback (most recent call last):
  ...
binarytree.exceptions.NodeModifyError: cannot delete the root node
```

```
>>> from binarytree import Node
>>>
>>> root = Node(1)           # index: 0, value: 1
>>> root.left = Node(2)      # index: 1, value: 2
>>> root.right = Node(3)     # index: 2, value: 3
>>>
```

(continues on next page)

(continued from previous page)

```
>>> del root[2]
>>>
>>> root[2]
Traceback (most recent call last):
  ...
binarytree.exceptions.NodeNotFoundError: node missing at index 2
```

__getitem__(*index: int*) → binarytree.Node
Return the node (or subtree) at the given level-order index.

Parameters **index** (*int*) – Level-order index of the node.

Returns Node (or subtree) at the given index.

Return type *binarytree.Node*

Raises

- ***binarytree.exceptions.NodeIndexError*** – If node index is invalid.
- ***binarytree.exceptions.NodeNotFoundError*** – If the node is missing.

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)        # index: 0, value: 1
>>> root.left = Node(2)   # index: 1, value: 2
>>> root.right = Node(3)  # index: 2, value: 3
>>>
>>> root[0]
Node(1)
>>> root[1]
Node(2)
>>> root[2]
Node(3)
>>> root[3]
Traceback (most recent call last):
  ...
binarytree.exceptions.NodeNotFoundError: node missing at index 3
```

__iter__() → Iterator[binarytree.Node]
Iterate through the nodes in the binary tree in level-order.

Returns Node iterator.

Return type Iterator[*binarytree.Node*]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)
```

(continues on next page)

(continued from previous page)

```
    __1
   /   \
  2     3
 / \
4   5

>>> list(root)
[Node(1), Node(2), Node(3), Node(4), Node(5)]
```

__len__() → int

Return the total number of nodes in the binary tree.

Returns Total number of nodes.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>>
>>> len(root)
3
```

Note: This method is equivalent to *binarytree.Node.size*.

__setitem__(*index: int*, *node: binarytree.Node*) → None

Insert a node (or subtree) at the given level-order index.

- An exception is raised if the parent node is missing.
- Any existing node or subtree is overwritten.
- Root node (current node) cannot be replaced.

Parameters

- **index** (*int*) – Level-order index of the node.
- **node** (binarytree.Node) – Node to insert.

Raises

- ***binarytree.exceptions.NodeTypeError*** – If new node is not an instance of *binarytree.Node*.
- ***binarytree.exceptions.NodeNotFoundError*** – If parent is missing.
- ***binarytree.exceptions.NodeModifyError*** – If user attempts to overwrite the root node (current node).

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)        # index: 0, value: 1
>>> root.left = Node(2)   # index: 1, value: 2
>>> root.right = Node(3)  # index: 2, value: 3
>>>
>>> root[0] = Node(4)
Traceback (most recent call last):
  ...
binarytree.exceptions.NodeModifyError: cannot modify the root node
```

```
>>> from binarytree import Node
>>>
>>> root = Node(1)        # index: 0, value: 1
>>> root.left = Node(2)   # index: 1, value: 2
>>> root.right = Node(3)  # index: 2, value: 3
>>>
>>> root[11] = Node(4)
Traceback (most recent call last):
  ...
binarytree.exceptions.NodeNotFoundError: parent node missing at index 5
```

```
>>> from binarytree import Node
>>>
>>> root = Node(1)        # index: 0, value: 1
>>> root.left = Node(2)   # index: 1, value: 2
>>> root.right = Node(3)  # index: 2, value: 3
>>>
>>> root[1] = Node(4)
>>>
>>> root.left
Node(4)
```

__str__() → str

Return the pretty-print string for the binary tree.

Returns Pretty-print string.

Return type str

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> print(root)

  __1
 /   \
2     3
 \
  4
```

Note: To include level-order indexes in the output string, use *binarytree.Node.pprint()* instead.

_repr_svg_() → str
Display the binary tree using Graphviz (used for Jupyter notebooks).

clone() → binarytree.Node
Return a clone of this binary tree.

Returns Root of the clone.

Return type *binarytree.Node*

equals(*other: binarytree.Node*) → bool
Check if this binary tree is equal to other binary tree.

Parameters **other** (binarytree.Node) – Root of the other binary tree.

Returns True if the binary trees are equal, False otherwise.

Return type bool

graphviz(**args*, ***kwargs*) → graphviz.graphs.Digraph
Return a graphviz.Digraph object representing the binary tree.

This method's positional and keyword arguments are passed directly into the Digraph's **__init__** method.

Returns graphviz.Digraph object representing the binary tree.

Raises **binarytree.exceptions.GraphvizImportError** – If graphviz is not installed

```
>>> from binarytree import tree
>>>
>>> t = tree()
>>>
>>> graph = t.graphviz()     # Generate a graphviz object
>>> graph.body               # Get the DOT body
>>> graph.render()           # Render the graph
```

height
Return the height of the binary tree.

Height of a binary tree is the number of edges on the longest path between the root node and a leaf node. Binary tree with just a single node has a height of 0.

Returns Height of the binary tree.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.left.left = Node(3)
>>>
>>> print(root)

    1
   /
```

(continues on next page)

(continued from previous page)

```
  2
 /
3

>>> root.height
2
```

Note: A binary tree with only a root node has a height of 0.

inorder

Return the nodes in the binary tree using in-order traversal.

An in-order traversal visits left subtree, root, then right subtree.

Returns List of nodes.

Return type [*binarytree.Node*]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.inorder
[Node(4), Node(2), Node(5), Node(1), Node(3)]
```

is_balanced

Check if the binary tree is height-balanced.

A binary tree is height-balanced if it meets the following criteria:

- Left subtree is height-balanced.
- Right subtree is height-balanced.
- The difference between heights of left and right subtrees is no more than 1.
- An empty binary tree is always height-balanced.

Returns True if the binary tree is balanced, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.left.left = Node(3)
>>>
>>> print(root)

    1
   /
  2
 /
3

>>> root.is_balanced
False
```

is_bst

Check if the binary tree is a BST (binary search tree).

Returns True if the binary tree is a BST, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(2)
>>> root.left = Node(1)
>>> root.right = Node(3)
>>>
>>> print(root)

  2
 / \
1   3

>>> root.is_bst
True
```

is_complete

Check if the binary tree is complete.

A binary tree is complete if it meets the following criteria:

- All levels except possibly the last are completely filled.
- Last level is left-justified.

Returns True if the binary tree is complete, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
```

(continues on next page)

(continued from previous page)

```
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.is_complete
True
```

is_max_heap

Check if the binary tree is a max heap.

Returns True if the binary tree is a max heap, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(3)
>>> root.left = Node(1)
>>> root.right = Node(2)
>>>
>>> print(root)

  3
 / \
1   2

>>> root.is_max_heap
True
```

is_min_heap

Check if the binary tree is a min heap.

Returns True if the binary tree is a min heap, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>>
>>> print(root)

  1
```

(continues on next page)

(continued from previous page)

```
 / \
2   3

>>> root.is_min_heap
True
```

is_perfect

Check if the binary tree is perfect.

A binary tree is perfect if all its levels are completely filled. See example below for an illustration.

Returns True if the binary tree is perfect, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>> root.right.left = Node(6)
>>> root.right.right = Node(7)
>>>
>>> print(root)

    __1__
   /     \
  2       3
 / \     / \
4   5   6   7

>>> root.is_perfect
True
```

is_strict

Check if the binary tree is strict.

A binary tree is strict if all its non-leaf nodes have both the left and right child nodes.

Returns True if the binary tree is strict, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)
```

(continues on next page)

(continued from previous page)

```
    __1
   /   \
  2     3
 / \
4   5

>>> root.is_strict
True
```

is_symmetric

Check if the binary tree is symmetric.

A binary tree is symmetric if it meets the following criteria:

- Left subtree is a mirror of the right subtree about the root node.

Returns True if the binary tree is a symmetric, False otherwise.

Return type bool

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(2)
>>> root.left.left = Node(3)
>>> root.left.right = Node(4)
>>> root.right.left = Node(4)
>>> root.right.right = Node(3)
>>>
>>> print(root)

    __1__
   /     \
  2       2
 / \     / \
3   4   4   3

>>> root.is_symmetric
True
```

leaf_count

Return the total number of leaf nodes in the binary tree.

A leaf node is a node with no child nodes.

Returns Total number of leaf nodes.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
```

(continues on next page)

(continued from previous page)

```
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> root.leaf_count
2
```

leaves

Return the leaf nodes of the binary tree.

A leaf node is any node that does not have child nodes.

Returns List of leaf nodes.

Return type [*binarytree.Node*]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> print(root)

  __1
 /   \
2     3
 \
  4

>>> root.leaves
[Node(3), Node(4)]
```

levelorder

Return the nodes in the binary tree using level-order traversal.

A level-order traversal visits nodes left to right, level by level.

Returns List of nodes.

Return type [*binarytree.Node*]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
```

(continues on next page)

(continued from previous page)

```
 / \
4   5

>>> root.levelorder
[Node(1), Node(2), Node(3), Node(4), Node(5)]
```

levels

Return the nodes in the binary tree level by level.

Returns Lists of nodes level by level.

Return type [[*binarytree.Node*]]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> print(root)

  __1
 /   \
2     3
 \
  4

>>>
>>> root.levels
[[Node(1)], [Node(2), Node(3)], [Node(4)]]
```

max_leaf_depth

Return the maximum leaf node depth of the binary tree.

Returns Maximum leaf node depth.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.right.left = Node(4)
>>> root.right.left.left = Node(5)
>>>
>>> print(root)

  1_____
 /      \
2        3
        /
       4
```

(continues on next page)

(continued from previous page)

```
     /
    5

>>> root.max_leaf_depth
3
```

max_node_value

Return the maximum node value of the binary tree.

Returns Maximum node value.

Return type float | int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>>
>>> root.max_node_value
3
```

min_leaf_depth

Return the minimum leaf node depth of the binary tree.

Returns Minimum leaf node depth.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.right.left = Node(4)
>>> root.right.left.left = Node(5)
>>>
>>> print(root)

  1_____
 /      \
2        3
        /
       4
      /
     5

>>> root.min_leaf_depth
1
```

min_node_value

Return the minimum node value of the binary tree.

Returns Minimum node value.

Return type float | int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>>
>>> root.min_node_value
1
```

postorder

Return the nodes in the binary tree using post-order traversal.

A post-order traversal visits left subtree, right subtree, then root.

Returns List of nodes.

Return type [*binarytree.Node*]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.postorder
[Node(4), Node(5), Node(2), Node(3), Node(1)]
```

pprint (*index: bool = False*, *delimiter: str = '-'*) → None

Pretty-print the binary tree.

Parameters

- **index** (*bool*) – If set to True (default: False), display level-order indexes using the format: `{index}{delimiter}{value}`.
- **delimiter** (*str*) – Delimiter character between the node index and the node value (default: '-').

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)               # index: 0, value: 1
>>> root.left = Node(2)          # index: 1, value: 2
>>> root.right = Node(3)         # index: 2, value: 3
>>> root.left.right = Node(4)    # index: 4, value: 4
```

(continues on next page)

(continued from previous page)

```
>>>
>>> root.pprint()

  __1
 /   \
2     3
 \
  4

>>> root.pprint(index=True)      # Format: {index}-{value}

   _____0-1_
  /         \
1-2_        2-3
    \
    4-4
```

Note: If you do not need level-order indexes in the output string, use *binarytree.Node.__str__()* instead.

preorder

Return the nodes in the binary tree using pre-order traversal.

A pre-order traversal visits root, left subtree, then right subtree.

Returns List of nodes.

Return type [*binarytree.Node*]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>>
>>> print(root)

    __1
   /   \
  2     3
 / \
4   5

>>> root.preorder
[Node(1), Node(2), Node(4), Node(5), Node(3)]
```

properties

Return various properties of the binary tree.

Returns Binary tree properties.

Return type dict

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.left = Node(4)
>>> root.left.right = Node(5)
>>> props = root.properties
>>>
>>> props['height']         # equivalent to root.height
2
>>> props['size']           # equivalent to root.size
5
>>> props['max_leaf_depth'] # equivalent to root.max_leaf_depth
2
>>> props['min_leaf_depth'] # equivalent to root.min_leaf_depth
1
>>> props['max_node_value'] # equivalent to root.max_node_value
5
>>> props['min_node_value'] # equivalent to root.min_node_value
1
>>> props['leaf_count']     # equivalent to root.leaf_count
3
>>> props['is_balanced']    # equivalent to root.is_balanced
True
>>> props['is_bst']         # equivalent to root.is_bst
False
>>> props['is_complete']    # equivalent to root.is_complete
True
>>> props['is_symmetric']   # equivalent to root.is_symmetric
False
>>> props['is_max_heap']    # equivalent to root.is_max_heap
False
>>> props['is_min_heap']    # equivalent to root.is_min_heap
True
>>> props['is_perfect']     # equivalent to root.is_perfect
False
>>> props['is_strict']      # equivalent to root.is_strict
True
```

size

Return the total number of nodes in the binary tree.

Returns Total number of nodes.

Return type int

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> root.size
4
```

Note: This method is equivalent to *binarytree.Node.__len__()*.

svg(*node_radius: int = 16*) → str
Generate SVG XML.

Parameters **node_radius** (*int*) – Node radius in pixels (default: 16).

Returns Raw SVG XML.

Return type str

validate() → None
Check if the binary tree is malformed.

Raises

- ***binarytree.exceptions.NodeReferenceError*** – If there is a cyclic reference to a node in the binary tree.
- ***binarytree.exceptions.NodeTypeError*** – If a node is not an instance of *binarytree.Node*.
- ***binarytree.exceptions.NodeValueError*** – If node value is invalid.

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = root  # Cyclic reference to root
>>>
>>> root.validate()
Traceback (most recent call last):
 ...
binarytree.exceptions.NodeReferenceError: cyclic node reference at index 0
```

values
Return the list representation of the binary tree.

Returns List representation of the binary tree, which is a list of node values in breadth-first order starting from the root. If a node is at index i, its left child is always at 2i + 1, right child at 2i + 2, and parent at index floor((i - 1) / 2). None indicates absence of a node at that index. See example below for an illustration.

Return type [float | int | None]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.left.left = Node(3)
>>> root.left.left.left = Node(4)
>>> root.left.left.right = Node(5)
>>>
>>> root.values
[1, 2, None, 3, None, None, None, 4, 5]
```

values2
Return the list representation (version 2) of the binary tree.

Returns List of node values like those from *binarytree.Node.values()*, but with a slightly different representation which associates two adjacent child values with the first parent value that has not been associated yet. This representation does not provide the same indexing properties where if a node is at index i, its left child is always at 2i + 1, right child at 2i + 2, and parent at floor((i - 1) / 2), but it allows for more compact lists as it does not hold "None"s between nodes in each level. See example below for an illustration.

Return type [float | int | None]

Example:

```
>>> from binarytree import Node
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.left.left = Node(3)
>>> root.left.left.left = Node(4)
>>> root.left.left.right = Node(5)
>>>
>>> root.values
[1, 2, None, 3, None, None, None, 4, 5]
>>> root.values2
[1, 2, None, 3, None, 4, 5]
```

3.2.2 Function: binarytree.build

binarytree.**build**(*values: Union[List[Optional[float]], List[Optional[int]], List[Optional[str]], List[float], List[int], List[str]]*) → Optional[binarytree.Node]
Build a tree from list representation and return its root node.

Parameters **values** (*[float | int | str | None]*) – List representation of the binary tree, which is a list of node values in breadth-first order starting from the root (current node). If a node is at index i, its left child is always at 2i + 1, right child at 2i + 2, and parent at floor((i - 1) / 2). "None" indicates absence of a node at that index. See example below for an illustration.

Returns Root node of the binary tree.

Return type binarytree.Node | None

Raises ***binarytree.exceptions.NodeNotFoundError*** – If the list representation is malformed (e.g. a parent node is missing).

Example:

```
>>> from binarytree import build
>>>
>>> root = build([1, 2, 3, None, 4])
>>>
>>> print(root)

  __1
 /   \
2     3
 \
  4
```

```
>>> from binarytree import build
>>>
>>> root = build([None, 2, 3])
Traceback (most recent call last):
  ...
binarytree.exceptions.NodeNotFoundError: parent node missing at index 0
```

3.2.3 Function: binarytree.build2

binarytree.**build2**(*values: List[Any]*) → Optional[binarytree.Node]

Build a tree from a list of values and return its root node.

Parameters **values** (*[float | int | str | None]*) – List of node values like those for *binarytree.build()*, but with a slightly different representation which associates two adjacent child values with the first parent value that has not been associated yet. This representation does not provide the same indexing properties where if a node is at index i, its left child is always at 2i + 1, right child at 2i + 2, and parent at floor((i - 1) / 2), but it allows for more compact lists as it does not hold "None"s between nodes in each level. See example below for an illustration.

Returns Root node of the binary tree.

Return type binarytree.Node | None

Raises ***binarytree.exceptions.NodeNotFoundError*** – If the list representation is malformed (e.g. a parent node is missing).

Example:

```
>>> from binarytree import build2
>>>
>>> root = build2([2, 5, None, 3, None, 1, 4])
>>>
>>> print(root)

      2
     /
  __5
 /
3
 / \
1   4
```

```
>>> from binarytree import build2
>>>
>>> root = build2([None, 1, 2])
Traceback (most recent call last):
  ...
binarytree.exceptions.NodeValueError: node value must be a float/int/str
```

3.2.4 Function: binarytree.tree

binarytree.**tree**(*height: int = 3*, *is_perfect: bool = False*, *letters: bool = False*) → Optional[binarytree.Node]

Generate a random binary tree and return its root node.

Parameters

- **height** (*int*) – Height of the tree (default: 3, range: 0 - 9 inclusive).
- **is_perfect** (*bool*) – If set to True (default: False), a perfect binary tree with all levels filled is returned. If set to False, a perfect binary tree may still be generated by chance.
- **letters** (*bool*) – If set to True (default: False), uppercase alphabet letters are used for node values instead of numbers.

Returns Root node of the binary tree.

Return type *binarytree.Node*

Raises ***binarytree.exceptions.TreeHeightError*** – If height is invalid.

Example:

```
>>> from binarytree import tree
>>>
>>> root = tree()
>>>
>>> root.height
3
```

```
>>> from binarytree import tree
>>>
>>> root = tree(height=5, is_perfect=True)
>>>
>>> root.height
5
>>> root.is_perfect
True
```

```
>>> from binarytree import tree
>>>
>>> root = tree(height=20)
Traceback (most recent call last):
  ...
binarytree.exceptions.TreeHeightError: height must be an int between 0 - 9
```

3.2.5 Function: binarytree.bst

binarytree.**bst**(*height: int = 3*, *is_perfect: bool = False*, *letters: bool = False*) → Optional[binarytree.Node]

Generate a random BST (binary search tree) and return its root node.

Parameters

- **height** (*int*) – Height of the BST (default: 3, range: 0 - 9 inclusive).
- **is_perfect** (*bool*) – If set to True (default: False), a perfect BST with all levels filled is returned. If set to False, a perfect BST may still be generated by chance.
- **letters** (*bool*) – If set to True (default: False), uppercase alphabet letters are used for node values instead of numbers.

Returns Root node of the BST.

Return type *binarytree.Node*

Raises ***binarytree.exceptions.TreeHeightError*** – If height is invalid.

Example:

```
>>> from binarytree import bst
>>>
>>> root = bst()
>>>
>>> root.height
3
>>> root.is_bst
True
```

```
>>> from binarytree import bst
>>>
>>> root = bst(10)
Traceback (most recent call last):
  ...
binarytree.exceptions.TreeHeightError: height must be an int between 0 - 9
```

3.2.6 Function: binarytree.heap

binarytree.**heap**(*height: int = 3*, *is_max: bool = True*, *is_perfect: bool = False*, *letters: bool = False*) → Optional[binarytree.Node]

Generate a random heap and return its root node.

Parameters

- **height** (*int*) – Height of the heap (default: 3, range: 0 - 9 inclusive).
- **is_max** (*bool*) – If set to True (default: True), generate a max heap. If set to False, generate a min heap. A binary tree with only the root node is considered both a min and max heap.
- **is_perfect** (*bool*) – If set to True (default: False), a perfect heap with all levels filled is returned. If set to False, a perfect heap may still be generated by chance.
- **letters** (*bool*) – If set to True (default: False), uppercase alphabet letters are used for node values instead of numbers.

Returns Root node of the heap.

Return type *binarytree.Node*

Raises ***binarytree.exceptions.TreeHeightError*** – If height is invalid.

Example:

```
>>> from binarytree import heap
>>>
>>> root = heap()
>>>
>>> root.height
3
>>> root.is_max_heap
True
```

```
>>> from binarytree import heap
>>>
>>> root = heap(4, is_max=False)
```

(continues on next page)

(continued from previous page)

```
>>>
>>> root.height
4
>>> root.is_min_heap
True
```

```
>>> from binarytree import heap
>>>
>>> root = heap(5, is_max=False, is_perfect=True)
>>>
>>> root.height
5
>>> root.is_min_heap
True
>>> root.is_perfect
True
```

```
>>> from binarytree import heap
>>>
>>> root = heap(-1)
Traceback (most recent call last):
 ...
binarytree.exceptions.TreeHeightError: height must be an int between 0 - 9
```

3.2.7 Function: binarytree.get_index

binarytree.**get_index**(*root: binarytree.Node*, *descendent: binarytree.Node*) → int

Return the level-order index given the root and a possible descendent.

Returns Level-order index of the descendent relative to the root node.

Return type int

Raises

- ***binarytree.exceptions.NodeTypeError*** – If root or descendent is not an instance of *binarytree.Node*.
- ***binarytree.exceptions.NodeReferenceError*** – If given a node that is not a root/descendent.

Example:

```
>>> from binarytree import Node, get_index
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> get_index(root, root.left)
1
>>> get_index(root, root.right)
2
>>> get_index(root, root.left.right)
4
```

(continues on next page)

(continued from previous page)

```
>>> get_index(root.left, root.right)
Traceback (most recent call last):
  ...
binarytree.exceptions.NodeReferenceError: given nodes are not in the same tree
```

3.2.8 Function: binarytree.get_parent

binarytree.**get_parent**(*root: Optional[binarytree.Node]*, *child: Optional[binarytree.Node]*) → Optional[binarytree.Node]

Search the binary tree and return the parent of given child.

Parameters

- **root** – Root node of the binary tree.
- **child** – Child node.

Type binarytree.Node | None

Return type binarytree.Node | None

Returns Parent node, or None if missing.

Return type binarytree.Node | None

Example:

```
>>> from binarytree import Node, get_parent
>>>
>>> root = Node(1)
>>> root.left = Node(2)
>>> root.right = Node(3)
>>> root.left.right = Node(4)
>>>
>>> print(root)

  __1
 /   \
2     3
 \
  4

>>> print(get_parent(root, root.left.right))

2
 \
  4
```

3.3 Exceptions

This page contains exceptions raised by **binarytree**:

exception binarytree.exceptions.**BinaryTreeError**

Base (catch-all) binarytree exception.

exception binarytree.exceptions.**NodeIndexError**

Node index was invalid.

exception binarytree.exceptions.**NodeModifyError**
User tried to overwrite or delete the root node.

exception binarytree.exceptions.**NodeNotFoundError**
Node was missing from the binary tree.

exception binarytree.exceptions.**NodeReferenceError**
Node reference was invalid (e.g. cyclic reference).

exception binarytree.exceptions.**NodeTypeError**
Node was not an instance of *binarytree.Node*.

exception binarytree.exceptions.**NodeValueError**
Node value was not a number (e.g. float, int, str).

exception binarytree.exceptions.**TreeHeightError**
Tree height was invalid.

3.4 Graphviz and Jupyter Notebook

From version 6.0.0, binarytree can integrate with Graphviz to render trees in image viewers, browsers and Jupyter notebooks using the python-graphviz library.

In order to use this feature, you must first install the Graphviz software in your OS and ensure its executables are on your PATH system variable (usually done automatically during installation):

```
# Ubuntu and Debian
$ sudo apt install graphviz

# Fedora and CentOS
$ sudo yum install graphviz

# Windows using choco (or winget)
$ choco install graphviz
```

Use *binarytree.Node.graphviz()* to generate graphviz.Digraph objects:

```
from binarytree import tree

t = tree()

# Generate a graphviz.Digraph object
# Arguments to this method are passed into Digraph.__init__
graph = t.graphviz()

# Get DOT (graph description language) body
graph.body

# Render the binary tree
graph.render()
```

With Graphviz you can also visualize binary trees in Jupyter notebooks:

Python Module Index

b

Index

Symbols

B

C

E

G

H

I

L

M

N

P

S

T

V