

[Log in](#)
[Subscribe RSS Feed](#)

Laurent Luce's Blog

Technical blog on web technologies

[Home](#)
[About](#)

Recent Posts

[Least frequently used cache eviction scheme with complexity \$O\(1\)\$ in Python](#)
[Cambridge city geospatial statistics](#)
[API to access the Cambridge city geospatial data](#)
[REST service + Python client to access geographic data](#)
[Massachusetts Census 2010 Towns maps and statistics using Python](#)
[Python, Twitter statistics and the 2012 French presidential election](#)
[Twitter sentiment analysis using Python and NLTK](#)
[Python dictionary implementation](#)
[Python string objects implementation](#)
[Python integer objects implementation](#)

Search

Meta

[Log in](#)
[Entries RSS](#)
[Comments RSS](#)
[WordPress.org](#)

Binary Search Tree library in Python

December 18, 2010

This article is about a Python library I created to manage binary search trees. I will go over the following:

- Node class
- Insert method
- Lookup method
- Delete method
- Print method
- Comparing 2 trees
- Generator returning the tree elements one by one

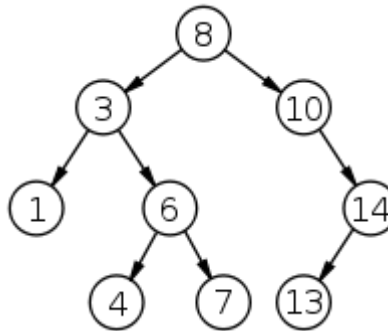
You can checkout the library code on GitHub: `git clone https://laurentluce@github.com/laurentluce/python-algorithms.git`. This folder contains more libraries but we are just going to focus on the Binary Tree one.

As a reminder, here is a binary search tree definition (Wikipedia).

A binary search tree (BST) or ordered binary tree is a node-based binary tree data structure which has the following properties:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Here is an example of a binary search tree:



Node class

We need to represent a tree node. To do that, we create a new class named Node with 3 attributes:

- Left node
- Right node
- Node's data (same as key in the definition above.)

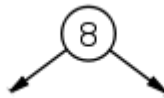
```

01 class Node:
02     """
03     Tree node: left and right child + data which can be any object
04     """
05     def __init__(self, data):
06         """
07         Node constructor
08
09         @param data node data object
10         """
11         self.left = None
12         self.right = None
13         self.data = data
  
```

Let's create a tree node containing the integer 8. You can pass any object for the data so it is flexible. When you create a node, both left and right node equal to None.

```
1 | root = Node(8)
```

Note that we just created a tree with a single node.



Insert method

We need a method to help us populate our tree. This method takes the node's data as an argument and inserts a new node in the tree.

```

01 class Node:
02     ...
03     def insert(self, data):
04         """
05         Insert new node with data
06
07         @param data node data object to insert
08         """
09         if self.data:
10             if data < self.data:
11                 if self.left is None:
12                     self.left = Node(data)
13                 else:
14                     self.left.insert(data)
15             elif data > self.data:
16                 if self.right is None:
17                     self.right = Node(data)
18                 else:
19                     self.right.insert(data)
20         else:
21             self.data = data

```

insert() is called recursively as we are locating the place where to add the new node.

Let's add 3 nodes to our root node which we created above and let's look at what the code does.

```

1 root.insert(3)
2 root.insert(10)
3 root.insert(1)

```

This is what happens when we add the second node (3):

- 1- root node's method insert() is called with data = 3.
- 2- 3 is less than 8 and left child is None so we attach the new node to it.

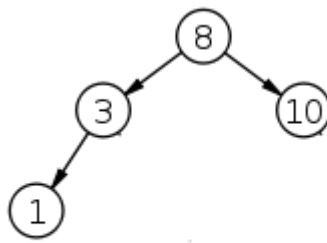
This is what happens when we add the third node (10):

- 1- root node's method insert() is called with data = 10.
- 2- 10 is greater than 8 and right child is None so we attach the new node to it.

This is what happens when we add the fourth node (1):

- 1- root node's method insert() is called with data = 1.
- 2- 1 is less than 8 so the root's left child (3) insert() method is called with data = 1. Note how we call the method on a subtree.
- 3- 1 is less than 3 and left child is None so we attach the new node to it.

This is how the tree looks like now:

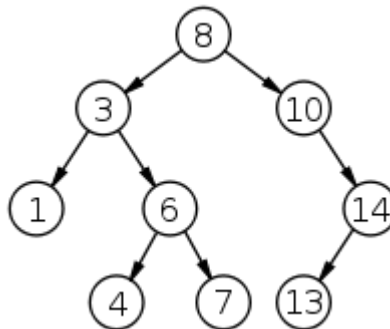


Let's continue and complete our tree so we can move on to the next section which is about looking up nodes in the tree.

```

1 | root.insert(6)
2 | root.insert(4)
3 | root.insert(7)
4 | root.insert(14)
5 | root.insert(13)
  
```

The complete tree looks like this:



Lookup method

We need a way to look for a specific node in the tree. We add a new method named lookup which takes a node's data as an argument and returns the node if found or None if not. We also return the node's parent for convenience.

```

01 | class Node:
02 |     ...
03 |     def lookup(self, data, parent=None):
04 |         """
05 |         Lookup node containing data
06 |
07 |         @param data node data object to look up
08 |         @param parent node's parent
09 |         @returns node and node's parent if found or None, None
10 |         """
11 |         if data < self.data:
12 |             if self.left is None:
13 |                 return None, None
14 |             return self.left.lookup(data, self)
15 |         elif data > self.data:
  
```

```
16         if self.right is None:
17             return None, None
18         return self.right.lookup(data, self)
19     else:
20         return self, parent
```

or right child to None. If it is the root node (no parent), we clear its data.

```

01 def delete(self, data):
02     ...
03     if children_count == 0:
04         # if node has no children, just remove it
05         if parent:
06             if parent.left is node:
07                 parent.left = None
08             else:
09                 parent.right = None
10             del node
11         else:
12             self.data = None
13     ...

```

Note: `children_count()` returns the number of children of a node.

Here is the function `children_count`:

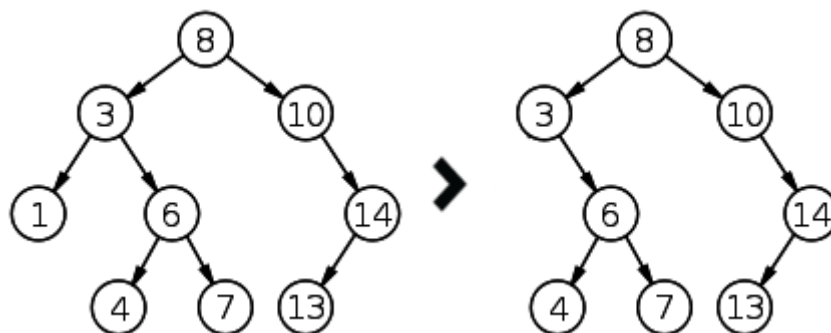
```

01 class Node:
02     ...
03     def children_count(self):
04         """
05         Returns the number of children
06
07         @returns number of children: 0, 1, 2
08         """
09         cnt = 0
10         if self.left:
11             cnt += 1
12         if self.right:
13             cnt += 1
14         return cnt

```

For example, we want to remove node 1. Node 3 left child will be set to None.

```
1 root.delete(1)
```



Let's look at the second possibility which is the node to be removed has one child. We replace the node with its child. We also handle the special case when the node is the root of the tree.

```

01 def delete(self, data):
02     ...
03     elif children_count == 1:
04         # if node has 1 child

```

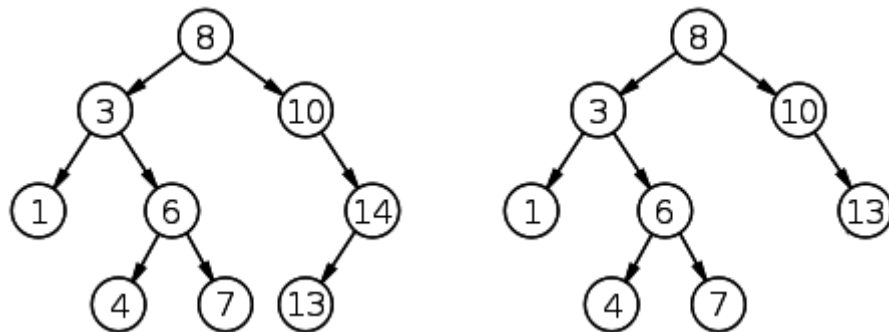
```

05 |         # replace node with its child
06 |         if node.left:
07 |             n = node.left
08 |         else:
09 |             n = node.right
10 |         if parent:
11 |             if parent.left is node:
12 |                 parent.left = n
13 |             else:
14 |                 parent.right = n
15 |             del node
16 |         else:
17 |             self.left = n.left
18 |             self.right = n.right
19 |             self.data = n.data
20 |     ...

```

For example, we want to remove node 14. Node 14 data will be set to 13 (its left child's data) and its left child will be set to None.

```
1 | root.delete(14)
```



Let's look at the last possibility which is the node to be removed has 2 children. We replace its data with its successor's data and we fix the successor's parent's child.

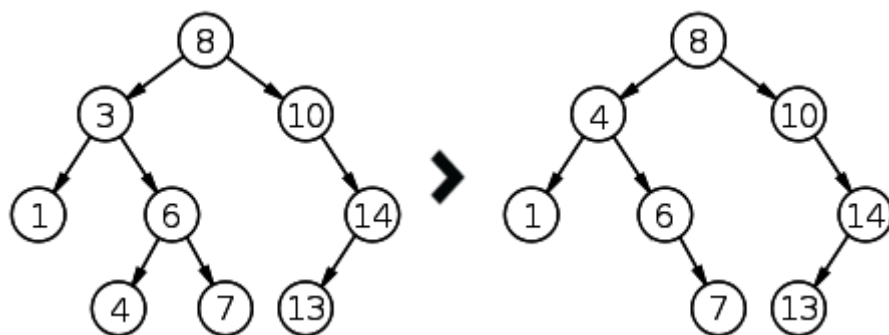
```

01 | def delete(self, data):
02 |     ...
03 |     else:
04 |         # if node has 2 children
05 |         # find its successor
06 |         parent = node
07 |         successor = node.right
08 |         while successor.left:
09 |             parent = successor
10 |             successor = successor.left
11 |         # replace node data by its successor data
12 |         node.data = successor.data
13 |         # fix successor's parent's child
14 |         if parent.left == successor:
15 |             parent.left = successor.right
16 |         else:
17 |             parent.right = successor.right

```

For example, we want to remove node 3. We look for its successor by going right then left until we reach a leaf. Its successor is node 4. We replace 3 with 4. Node 4 doesn't have a child so we set node 6 left child to None.

```
1 | root.delete(3)
```



Print method

We add a method to print the tree inorder. This method has no argument. We use recursion inside `print_tree()` to walk the tree depth-first. We first traverse the left subtree, then we print the root node then we traverse the right subtree.

```
01 | class Node:
02 |     ...
03 |     def print_tree(self):
04 |         """
05 |         Print tree content inorder
06 |         """
07 |         if self.left:
08 |             self.left.print_tree()
09 |         print self.data,
10 |         if self.right:
11 |             self.right.print_tree()
```

Let's print our tree:

```
1 | root.print_tree()
```

The output will be: 1, 3, 4, 6, 7, 8, 10, 13, 14

Comparing 2 trees

To compare 2 trees, we add a method which compares each subtree recursively. It returns False when one leaf is not the same in both trees. This includes 1 leaf missing in the other tree or the data is different. We need to pass the root of the tree to compare to as an argument.

```
01 | class Node:
02 |     ...
03 |     def compare_trees(self, node):
04 |         """
05 |         Compare 2 trees
06 |
07 |         @param node tree's root node to compare to
08 |         @returns True if the tree passed is identical to this tree
09 |         """
10 |         if node is None:
11 |             return False
12 |         if self.data != node.data:
```



```

13         return False
14     res = True
15     if self.left is None:
16         if node.left:
17             return False
18     else:
19         res = self.left.compare_trees(node.left)
20     if res is False:
21         return False
22     if self.right is None:
23         if node.right:
24             return False
25     else:
26         res = self.right.compare_trees(node.right)
27     return res

```

For example, we want to compare tree (3, 8, 10) with tree (3, 8, 11)



```

1 # root2 is the root of tree 2
2 root.compare_trees(root2)

```

This is what happens in the code when we call `compare_trees()`.

- 1- The root node `compare_trees()` method is called with the tree to compare root node.
- 2- The root node has a left child so we call the left child `compare_trees()` method.
- 3- The left subtree comparison will return True.
- 2- The root node has a right child so we call the right child `compare_trees()` method.
- 3- The right subtree comparison will return False because the data is different.
- 4- `compare_trees()` will return False.

Generator returning the tree elements one by one

It is sometimes useful to create a generator which returns the tree nodes values one by one. It is memory efficient as it doesn't have to build the full list of nodes right away. Each time we call this method, it returns the next node value.

To do that, we use the `yield` keyword which returns an object and stops right there so the function will continue from there next time the method is called.

We cannot use recursion in this case so we use a stack.

Here is the code:

```

01 class Node:
02     ...
03     def tree_data(self):
04         """
05         Generator to get the tree nodes data
06         """
07         # we use a stack to traverse the tree in a non-recursive way

```

```

08     stack = []
09     node = self
10     while stack or node:
11         if node:
12             stack.append(node)
13             node = node.left
14         else: # we are returning so we pop the node and we yield it
15             node = stack.pop()
16             yield node.data
17             node = node.right

```

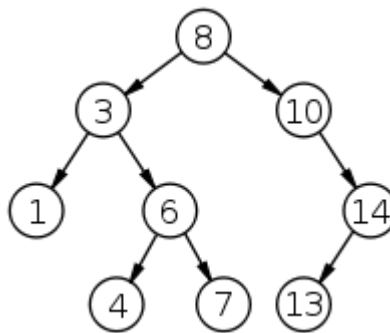
For example, we want to access the tree nodes using a for loop:

```

1 for data in root.tree_data():
2     print data

```

Let's look at what happens in the code with the same example we have been using:



- 1- The root node tree_data() method is called.
- 2- Node 8 is added to the stack. We go to the left child of 8.
- 3- Node 3 is added to the stack. We go to the left child of 3.
- 4- Node 1 is added to the stack. Node is set to None because there is no left child.
- 5- We pop a node which is Node 1. We yield it (returns 1 and stops here until tree_data() is called again).
- 6- tree_data() is called again because we are using it in a for loop.
- 7- Node is set to None because Node 1 doesn't have a right child.
- 8- We pop a node which is Node 3. We yield it (returns 3 and stops here until tree_data() is called again).
- ...

Here you go, I hope you enjoyed this tutorial. Don't hesitate to add comments if you have any feedback.

tags: [Python](#)

posted in [Uncategorized](#) by Laurent Luce

Follow comments via the [RSS Feed](#) | [Leave a comment](#) | [Trackback URL](#)

40 Comments to "Binary Search Tree library in Python"

1.



Ujjwol wrote:

Great! Help Thanks.

[Link](#) | December 19th, 2010 at 3:18 am

2.

[Nicolas Dumazet](#) wrote:

Hello!

First of all, let me say that this is a great article. You've obviously spent some time on formatting, graphs and code. Kudos.

But I wonder: is a BinaryTree really different from a Node?
Think about it. Change the Node constructor signature to `__init__(self, data=None)`, add a `getRoot`: `return self`

Isn't it the same?

It would allow you to define `insert/delete/lookup` on the BinaryTree object, as a method. Instead of using "root" as a first argument, you'd use "self".
`bt.insert(8)` instead of `bt_insert(root, 8)`

[Link](#) | December 20th, 2010 at 2:47 am

3.

*laurent* wrote:

@Nicolas Dumazet: Thanks for your feedback. I really like your suggestion of just using the class Node as it simplifies the methods arguments. I took the opportunity to update the tutorial with this new design.
Let me know what you think of it.

[Link](#) | December 20th, 2010 at 6:07 pm

4.

[Nicolas Dumazet](#) wrote:

You're welcome. I'm glad you liked the suggestion! \o/

Double-check the `children_count` function as well. It could be directly a method of Node instead of acting like a class method: `node.children_count()` instead of `self.children_count(node)`

Is `which_child` required? You can probably go away with an equality comparison. See <http://pastebin.com/VRwvN7aX> as an illustration.

Cheers!

[Link](#) | December 21st, 2010 at 1:46 am

5.

*laurent* wrote:

Thanks for your feedback. I updated the code.

[Link](#) | December 21st, 2010 at 12:53 pm

6.

*LE* wrote:

Interesting, one comment is that you should not compare to None, you should test for identity – i.e. "if x is None:" – comparison is the same as "if not x:" which makes it more clear what you're after (trueness or

identity)

[Link](#) | December 25th, 2010 at 9:24 am

7.



hamad mohammad wrote:

i found your explanation by steps is very helpfull for some one like my who can't get it from first time
thank you

and please continue i love your writing style

[Link](#) | February 11th, 2011 at 5:25 pm

8.



Laurent Luce wrote:

@hamad: Thanks.

[Link](#) | February 11th, 2011 at 11:55 pm

9.



hamad mohammad wrote:

if you elaborate little bit in the part of ((removing 2 children|))

[Link](#) | March 1st, 2011 at 9:53 am

10.



[Cristi Constantin](#) wrote:

Excellent tutorial !
It would be great if you add a json dump function. 😊

[Link](#) | May 16th, 2011 at 5:48 am

11.



[Adrian Lienhard](#) wrote:

Thanks for the post!

Note: there's a bug in the delete function – in the case of one child, only the data is moved up from the successor, but not the successor's children. Hence you'll loose a whole subtree.

[Link](#) | June 4th, 2011 at 1:23 pm

12.



Laurent Luce wrote:

@Adrian: I fixed that. Thanks!

[Link](#) | June 11th, 2011 at 2:43 pm

13.



nova wrote:

the method is located `lokoop` not work for me? please do some more detailed examples

[Link](#) | July 3rd, 2011 at 12:13 am

14.



Laurent Luce wrote:

@nova: What is not working? Please post more details and some code so I can help you.

[Link](#) | July 9th, 2011 at 4:00 pm

15.



game0n wrote:

```
# fix successor's parent's child
if parent.left == successor:
    parent.left = successor.right
else:
    parent.right = successor.right
```

for this, i don't understand a check is being made – we were traversing only to the left of parents to find the successor, right? so the code should have only `parent.left = successor.right`.

If that is not the case, can you please explain with example?
Thanks!

PS: your post has been by far the most helpful resource for me to understand the implementation.
Thanks!!

[Link](#) | March 15th, 2012 at 2:03 pm

16.



Jaimin wrote:

very good article!

[Link](#) | March 17th, 2012 at 1:51 pm

17.



[David Watson](#) wrote:

I found this article on binary tree in python very helpful. However, I noticed that the call to `tree_data` in the example above is missing the parentheses. I forked and sent you a pull request on github. I'm guessing it may confuse some readers. Cheers.

[Link](#) | April 10th, 2012 at 5:11 am

18.



Sarah wrote:

Hi,

I know this is from a year ago, but I am working on a project that requires a binary tree and I found this website so extremely helpful, especially the way you explained everything and made it more visual.

However, with my project my tree has to be letters (for a morse code translation) and I am having trouble with inserting the letters and creating the tree because every website and book I have found they are

inserting numbers. Do you have any suggestions on how this could be done? Any help is greatly appreciated.

[Link](#) | April 19th, 2012 at 2:41 pm

19.



Ben wrote:

@nova tree_data is missing the parens in the first example. You'll just need to add those to have it work.
i.e.

```
for data in root.tree_data():  
    print data
```

[Link](#) | May 11th, 2012 at 8:42 am

20.



Laurent Luce wrote:

@David Watson: Thanks!

[Link](#) | June 1st, 2012 at 6:31 pm

21.



Laurent Luce wrote:

@Sarah: data in the Node object can be anything. It can be a string like 'A', 'B'... Also, 'A' < 'B' is True in Python.

[Link](#) | June 1st, 2012 at 6:40 pm

22.



Laurent Luce wrote:

@gameOn: In the example where we remove Node 3. If Node 6 has no children then parent.right will be equal to successor.

[Link](#) | June 1st, 2012 at 6:58 pm

23.



[Manoj](#) wrote:

beautiful programming, I read only up till init, insert, find. while I read the c implementations on data structures by Weiss and was looking for translating it to python I had difficulties so I came over to learn from yours 😊

[Link](#) | July 4th, 2012 at 3:37 am

24.



[Manoj](#) wrote:

u can never delete root right? and so does a node deleted bubbled down by its successors until a leaf or 1 child is reached which is deleted easily???

[Link](#) | July 9th, 2012 at 6:51 am

25.

*Christo* wrote:

Great article. Extremely well written! Thanks for putting it together it has helped me a lot.

[Link](#) | July 16th, 2012 at 12:09 pm

26.

*snehaa* wrote:

excellent work... helps alot to understand the concepts of trees very clearly and deeply.. thanks alot...

[Link](#) | April 11th, 2013 at 9:54 am

27.

*Raveesh* wrote:

Awesome! Loved your detailed explanation of each and every step! 😊

[Link](#) | August 16th, 2013 at 5:45 pm

28.

*Uthpala* wrote:

very well written,nice and detailed article.Thank you very much.

[Link](#) | October 7th, 2013 at 9:27 pm

29.

*Chris* wrote:

Thanks for sharing, Laurent. I ended up using some of your code in my CS 101 class today:

<http://w3.cs.jmu.edu/mayfiecs/cs101/wk-10/Lab10-BinaryTree.html>

[Link](#) | November 1st, 2013 at 12:42 pm

30.

*Matt L* wrote:

Great article – a small suggestion is in the insert method to change “else:” to “elif data > self.data:” to throw out duplicates – it will then match the definition of binary search tree you’ve provided (and your lookup and delete methods will not have to handle multiple match cases).

[Link](#) | January 1st, 2014 at 5:26 pm

31.

*Dan Stromberg* wrote:

I’ve taken this page and packaged it up as a Python 2.x and 3.x Dictionary-like binary tree module. It’s at

<http://stromberg.dnsalias.org/~strombrg/linked-list/>

[Link](#) | February 10th, 2014 at 6:22 pm

32.

*Liangyue Li* wrote:

For comparing two binary trees,
`res = self.left.compare_trees(node.left)`
 if the two left subtrees are not the same, `res = false`, but will be overwritten by `res = self.right.compare_trees(node.right)`
 , which will return true if the two right subtrees are the same.

[Link](#) | March 21st, 2014 at 10:20 pm

33.



Laurent Luce wrote:

@Liangyue Li: Thanks! I updated the method to return False if `self.left.compare_trees` returns False.

[Link](#) | May 25th, 2014 at 8:48 pm

34.



Laurent Luce wrote:

@Matt L: Great suggestion!

[Link](#) | May 25th, 2014 at 9:17 pm

35.



Titus wrote:

Hey, great article!

One small note, in the `children_count` method the lines 9 and 10 are not needed.

```

    09
    if node is None:
    10
    return None

```

Cheers

[Link](#) | July 12th, 2014 at 10:57 am

36.



Laurent Luce wrote:

@Titus: Those lines were removed on GitHub but not on the post. Thanks!

[Link](#) | July 23rd, 2014 at 9:33 pm

37.



Barnabas wrote:

Hey, really nice article, found it really useful! Only one little thing, when you describe the print function, you've mistaken the in-order traversal to be breadth-first instead depth-first. Cheers!

[Link](#) | November 1st, 2014 at 5:30 am

38.



Laurent Luce wrote:

@Barnabas: Thanks, updated.

[Link](#) | December 25th, 2014 at 6:30 pm

39.



[David](#) wrote:

¡Hello!

I have been looking for something like this and didn't find anything for a long.

I'm a systems engineering student and never understood this trees stuff like i just did with your explanation, thank you so much!

[Link](#) | May 3rd, 2015 at 3:48 am

40.



[Dee](#) wrote:

Excellent tutorial, we are going through this in CS right now and this tutorial has taught me so much!
Thank you!!

[Link](#) | October 23rd, 2015 at 11:41 am

Leave Your Comment

Name (required)

Mail (will not be published) (required)

Website

Post Comment

Powered by [Wordpress](#) and [MySQL](#). Theme by [Shlomi Noach](#), [openark.org](#)