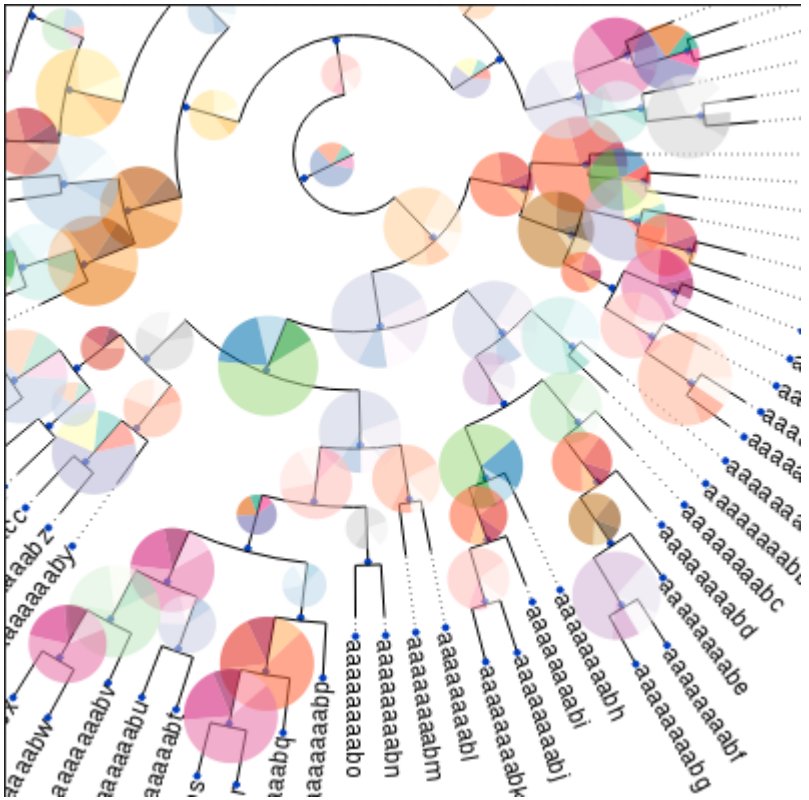
# ETE3 Tutorial

http://etetoolkit.org/




Marina Marcet-Houben
mmarcet@crg.es

# How to install ete3

## Linux + MAC OS X

Using Conda (no sudo required) | Native

1. Installing ETE using the Anaconda/Miniconda environment

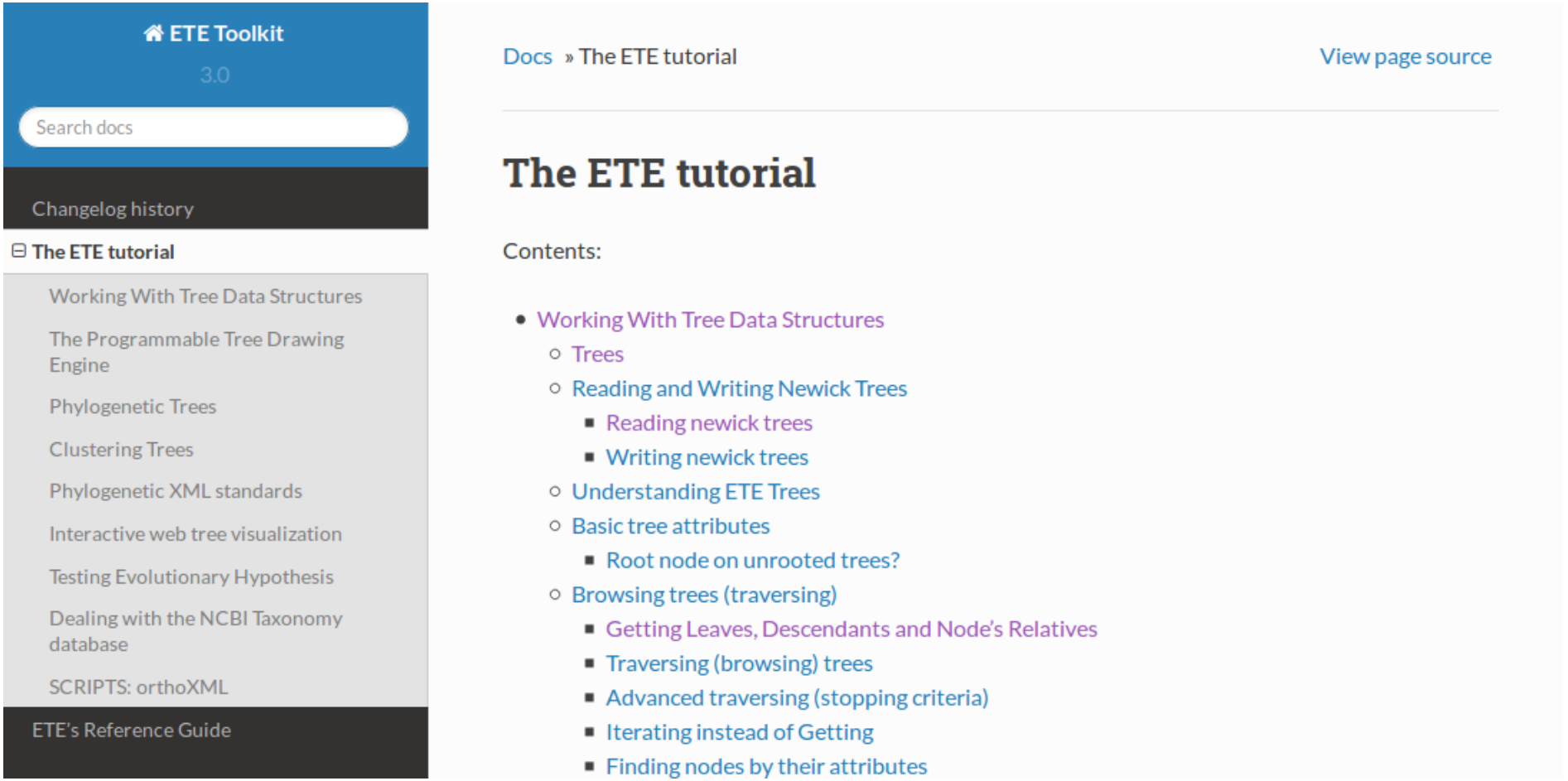
```
# Install Minconda  (you can ignore this step if you already have Anaconda/Miniconda)
wget http://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh -O Miniconda3-latest-Linux-x86_64.sh
bash Miniconda3-latest-Linux-x86_64.sh -b -p ~/anaconda_ete/
export PATH=~/anaconda_ete/bin:$PATH;

# Install ETE
conda install -c etetoolkit ete3 ete_toolchain

# Check installation
ete3 build check
```

2. Run the following line to activate Anaconda before using ETE:

```
# Activate the environment
export PATH=~/anaconda_ete/bin:$PATH
```

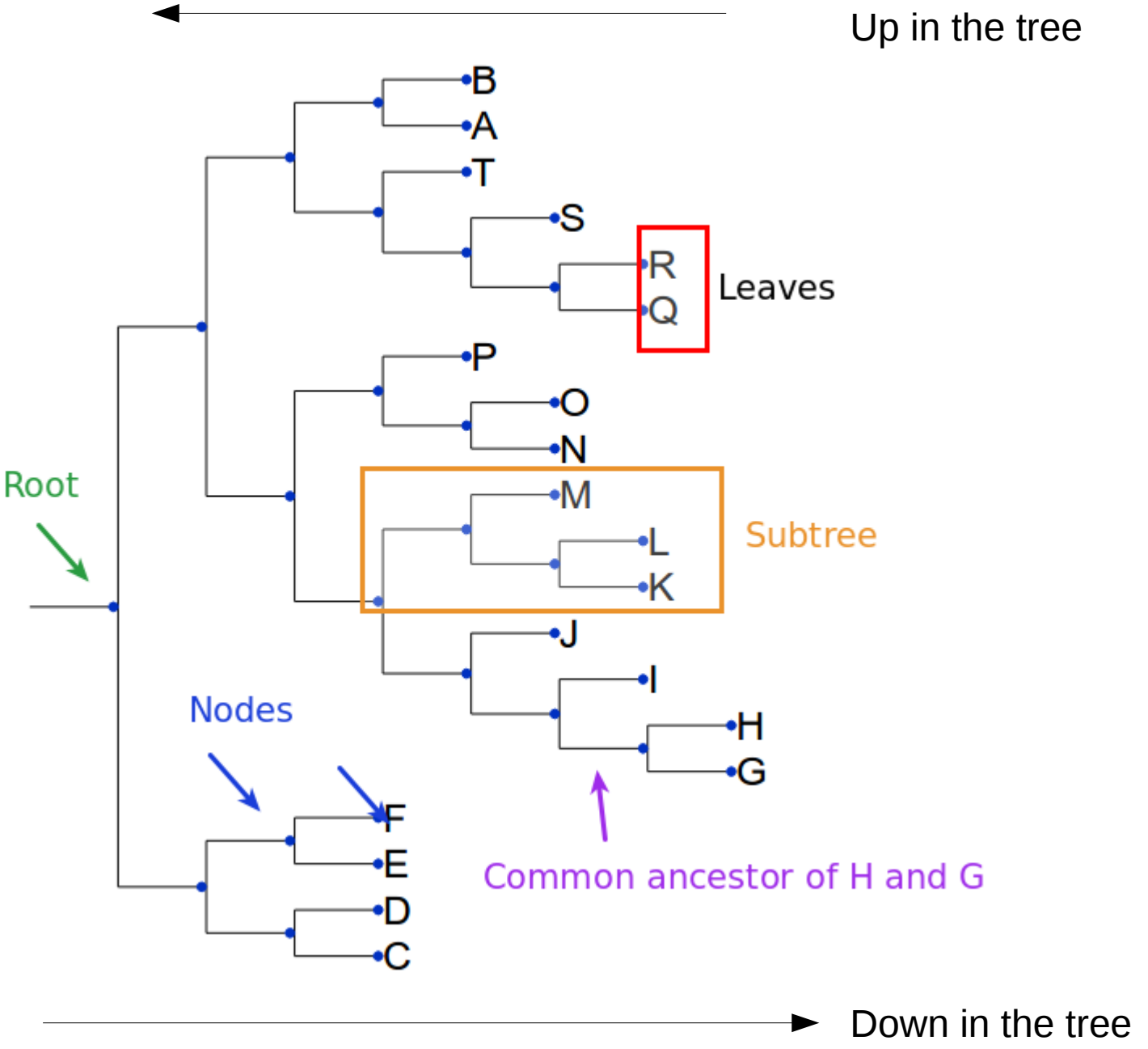
Tip: you can add the line to your `.bashrc` to make it permanent

**Conda rules!!!**

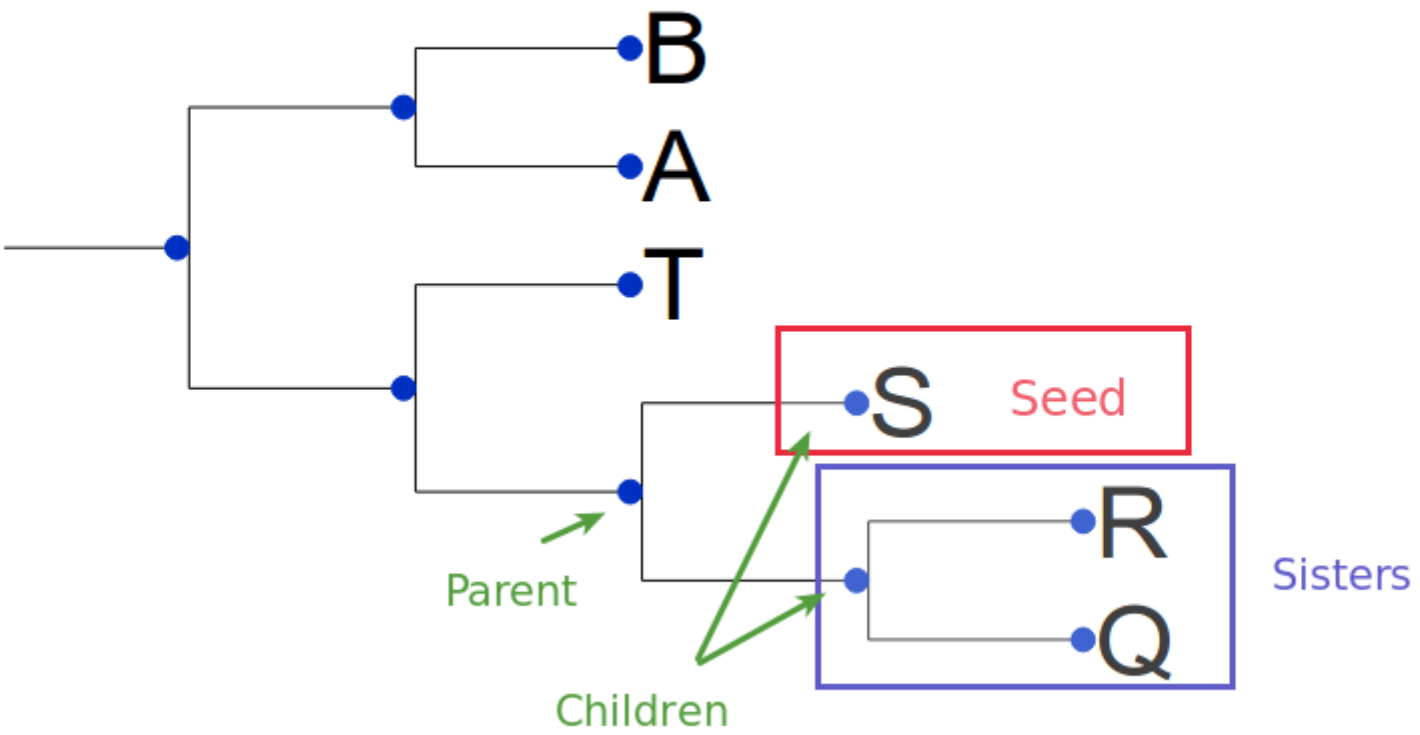



I will go through the most normal commands (i.e. the ones I usually use), but to cover the whole list it's better to refer to the tutorial.

Some terms so that everyone is on the same page



Some terms so that everyone is on the same page




We are going to use a lot of randomly generated trees for the examples, so don’t be worried if your results don’t exactly match the ones found in these slides!

Before starting to use ETE, whether in an interactive terminal or in a script, you need to import it:

```
import ete3
```

If you only need some parts of ete3 you could also import the specific part such as:

```
from ete3 import Tree
```

The difference is that in the first case you'll have to always start any command involving ete3 with ete3. Whereas in the second case you would directly write Tree.

```
ete3.Tree("((A,B),C);")

Tree("((A,B),C);"
```

***Notice that upper and lower cases are essential!!***

## First things first: What is a tree in ETE and which basic characteristics do trees have.

It does not need to be a phylogenetic tree, any hierarchical tree-like structure could in theory be used as long as it's in newick format.
So, in ETE, trees are just a bunch of interconnected nodes that form a hierarchical structure.

| Tree type | ete3.Tree() | ete3.PhyloTree() |
|---|---|---|
| Leaf names | Yes | Yes |
| Branch lengths | Yes | Yes |
| Support | Yes | Yes |
| Species information | No | Yes |

## How do I load trees in ETE

1.- Input a newick as a string:

```
t = ete3.Tree("((A,B),C);")
```

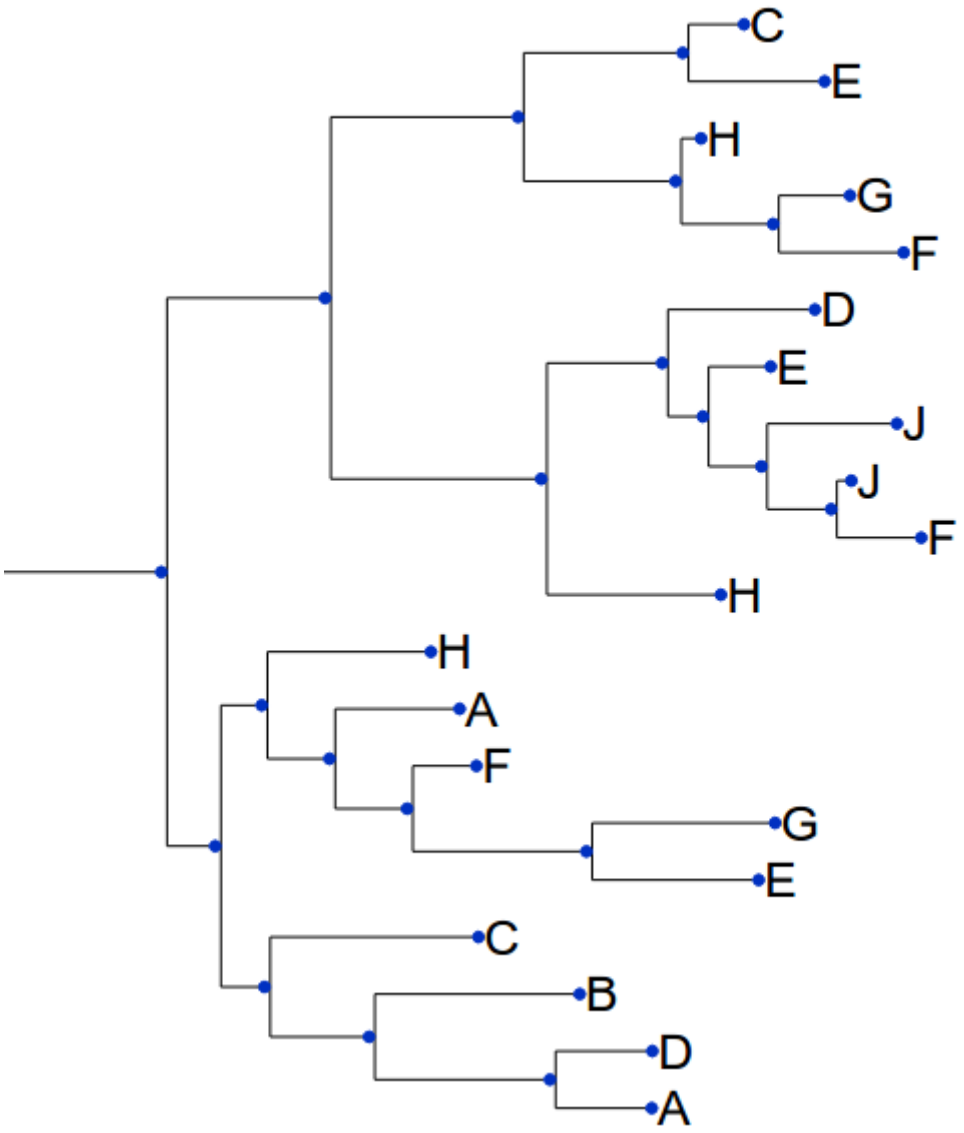
2.- Obtain a newick from a file (most common):

```
t = ete3.Tree("example_tree_1.txt")
```

3.- Create a random tree: it has options to modify the size of the tree, the leaf names, branch lengths and branch supports.

```
t = ete3.Tree()

t.populate(10)
```

```
t.populate(20,["A","B","C","D","E","F","G","H","I","J"],random_branches=True,reuse_names=True)
```



# How do I print trees in ETE

Load the example tree first:

```
t = ete3.Tree("example_tree_1.txt")
```

1.- Print on screen

```
print t
```


Notice that it only prints topology.

Writting only t will get you this:

```
t
Tree node '' (0x7fa1bb9764d)
```

2.- Write into a file:

```
t.write(outfile="example_tree_1.printed.txt")
```

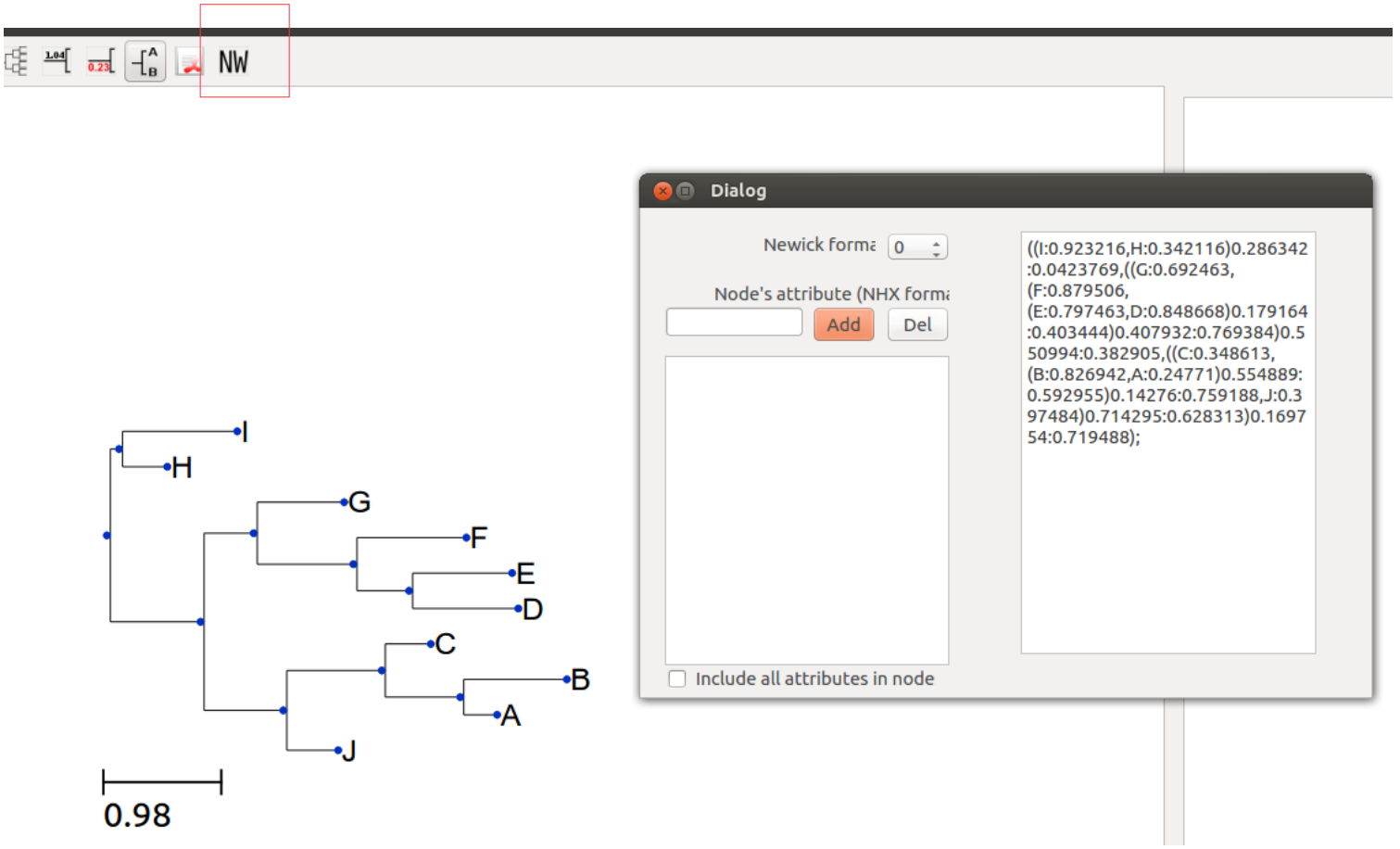
| FORMAT | DESCRIPTION | SAMPLE |
|---|---|---|
| 0 | flexible with support values | ((D:0.723274,F:0.567784)1.000000:0.067192,(B:0.279326,H:0.756049)1.000000:0.807788); |
| 1 | flexible with internal node names | ((D:0.723274,F:0.567784)E:0.067192,(B:0.279326,H:0.756049)B:0.807788); |
| 2 | all branches + leaf names + internal supports | ((D:0.723274,F:0.567784)1.000000:0.067192,(B:0.279326,H:0.756049)1.000000:0.807788); |
| 3 | all branches + all names | ((D:0.723274,F:0.567784)E:0.067192,(B:0.279326,H:0.756049)B:0.807788); |
| 4 | leaf branches + leaf names | ((D:0.723274,F:0.567784),(B:0.279326,H:0.756049)); |
| 5 | internal and leaf branches + leaf names | ((D:0.723274,F:0.567784):0.067192,(B:0.279326,H:0.756049):0.807788); |
| 6 | internal branches + leaf names | ((D,F):0.067192,(B,H):0.807788); |
| 7 | leaf branches + all names | ((D:0.723274,F:0.567784)E,(B:0.279326,H:0.756049)B); |
| 8 | all names | ((D,F)E,(B,H)B); |
| 9 | leaf names | ((D,F),(B,H)); |
| 100 | topology only | ((,),(,)); |

```
t.write(outfile="example_tree_1.printed9.txt",format=9)
```

You can also use the write format to print the tree on screen

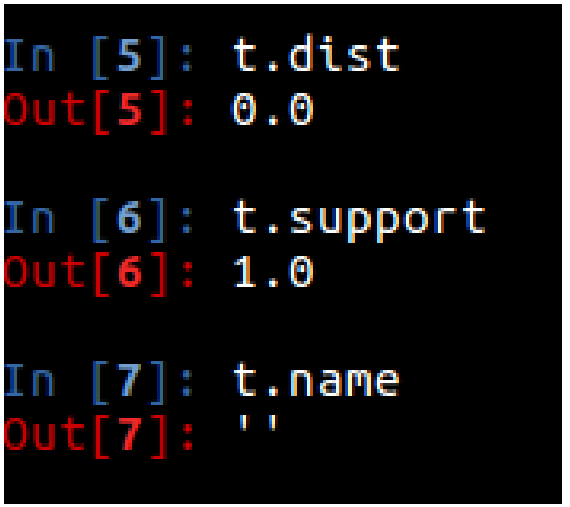
```
In [15]: t.write(format=9)
Out[15]: '((I,H),((G,(F,(E,D))),((C,(B,A)),J)));'

In [16]: t.write(format=1)
Out[16]: '((I:0.923216,H:0.342116):0.0423769,((G:0.692463,(F:0.879506,(E:0.79746
3,D:0.848668):0.403444):0.769384):0.382905,((C:0.348613,(B:0.826942,A:0.24771):0
.592955):0.759188,J:0.397484):0.628313):0.719488);'
```

3.- Get newick from the graphical interface



## Node attributes

All nodes in a tree structure have space for three attributes although they may be 0 depending on the tree. We can access them simply by:




Additional attributes can be added to a node, but we will explore this option later on.

## How to move through the tree structure

When you load a tree you will be always placed at the root of the tree. You can check that by typing:

```
t.is_root()
True
```

To go one node down in the tree structure, you need to call the children.

```
children = t.get_children()
```

For a rooted tree and for any other bifurcating nodes, there will be two children.

```
In [22]: len(children)
Out[22]: 2
```

```
In [23]: print children[0]

   /-I
--|
   \-H

In [24]: print children[1]

      /-G
   /-|
  |  |   /-F
  |   \-|
  |     |   /-E
  |      \-|
--|        \-D
  |
  |      /-C
  |   /-|
  |  |  |   /-B
   \-|   \-|
     |      \-A
     |
      \-J
```

You could now assign each child to a variable and search for their own children (grandchildren).

```
In [25]: ch1,ch2 = children

In [26]: grandchildren1 = ch1.get_children()

In [27]: grandchildren2 = ch2.get_children()
```

```
In [30]: print grandchildren1[0]

--I

In [31]: print grandchildren1[1]

--H
```

```
In [32]: print grandchildren2[0]

   /-G
--|
  |   /-F
   \-|
     |   /-E
      \-|
        \-D

In [33]: print grandchildren2[1]

      /-C
   /-|
  |  |   /-B
--|   \-|
  |      \-A
  |
   \-J
```

The grandchildren1 are now leaves, if we try to go farther down the tree it will return an empty list. To see whether one of the nodes is a leaf you can use:

```
In [35]: grandchildren1[1].is_leaf()
Out[35]: True
```

We can also get a leaf by its name

```
leaf = t.get_leaves_by_name("A")[0]
```

To move up in the tree hierarchy we can simply:

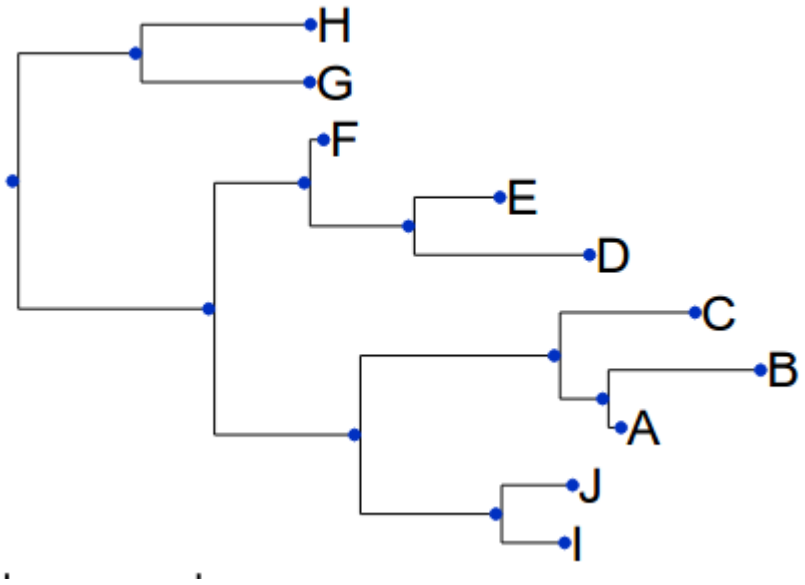
```
In [40]: parent = leaf.up

In [41]: print parent

   /-B
--|
   \-A
```

We can also go to the sister group:

```
In [44]: sister = leaf.get_sisters()[0]

In [45]: print sister

--B
```

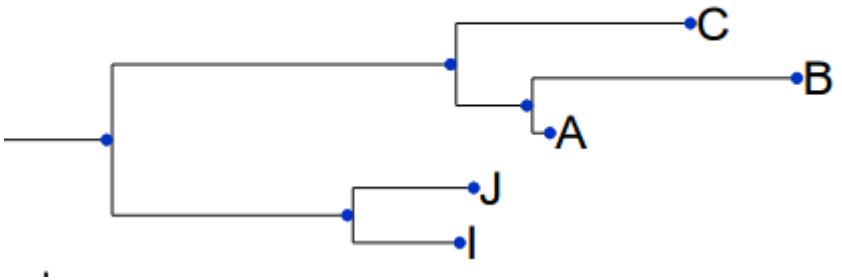
With these tree you can go to any place in the tree that you want, but it may not be very efficient if you want to cover the whole tree structure.

Finally we can search nodes that are the common ancestor of a set of leaves:




Original tree

```
anc = t.get_common_ancestor("A","B","I")
```

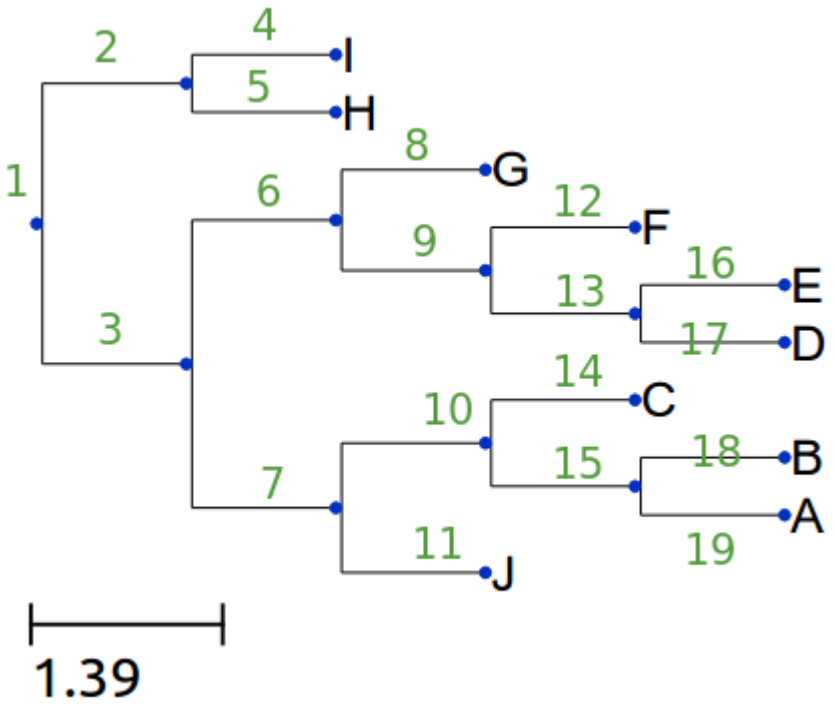



**WARNING: Getting the common ancestor of a single leaf will return the whole tree!!**

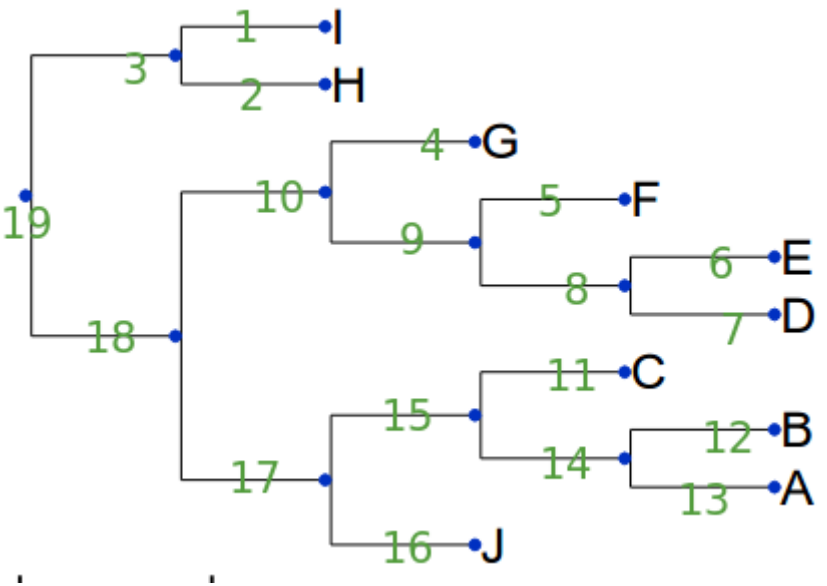
# Iter through the tree

The traverse function will allow you to go through each one of the tree nodes in a given order.

```
In [3]: num = 1

In [4]: for node in t.traverse():
   ...:     if node.is_leaf():
   ...:         print num,node.name
   ...:         num += 1
   ...:
1 I
2 H
3 G
4 J
5 F
6 C
7 E
8 D
9 B
10 A
```



```
In [10]: for node in t.traverse(strategy="postorder"):
    ....:     if node.is_leaf():
    ....:         print num,node.name
    ....:
1 I
1 H
1 G
1 F
1 E
1 D
1 C
1 B
1 A
1 J
```



## Iter through the leaves

If you only want to go through each leave, you can use these two options:

```
In [3]: for leaf in t.iter_leaves():
   ...:     print leaf
   ...:
```

You can also go through the leaf names:

```
In [4]: for leaf in t.iter_leaf_names():
   ...:     print leaf
   ...:
```

The first returns a node whereas the second returns only a string with the name.

The same options can be used with “get” in which case they will return a list: t.get_leaves(), t.get_leaf_names()

## Check whether a set of leaves are monophyletic

```
results = t.check_monophyly(["A","B","I"],"name")
```

This function outputs:

1.- Whether the leaf names provided form a monophyletic clade (values True / False)

2.- Which relationship they have (monophyletic or polyphyletic)

3.- The set of leaf names that break the monophyly.

## Add features to nodes

Additional information can be added to nodes by adding features. Imagine that in our example tree, we want to annotate whether the leaf is a vowel:

```
t = ete3.Tree("example_tree_1.txt")

for leaf in t.iter_leaves():
    if leaf.name == "A" or leaf.name == "I":
        leaf.add_feature("vowel",True)
    else:
        leaf.add_feature("vowel",False)
```

We can access this new feature in the same way we access the normal attributes.

```
In [60]: leaf.name
Out[60]: 'I'

In [61]: leaf.vowel
Out[61]: True
```

Once created an attribute can also be modified:

```
In [62]: leaf.vowel = "Something Else"

In [63]: leaf.vowel
Out[63]: 'Something Else'
```

You can also create multiple features at the same time

```
leaf.add_features(feature1=0,feature2=[])
```

To know which features are assigned to your node, you can simply:

```
In [66]: leaf.features
Out[66]: {'dist', 'feature1', 'feature2', 'name', 'support', 'vowel'}
```

## Tree comparison

ETE3 can be used to compare the topologies of two trees. It implements different measures of distance such as the Robinson Foulds distance.

Lets first get two trees:

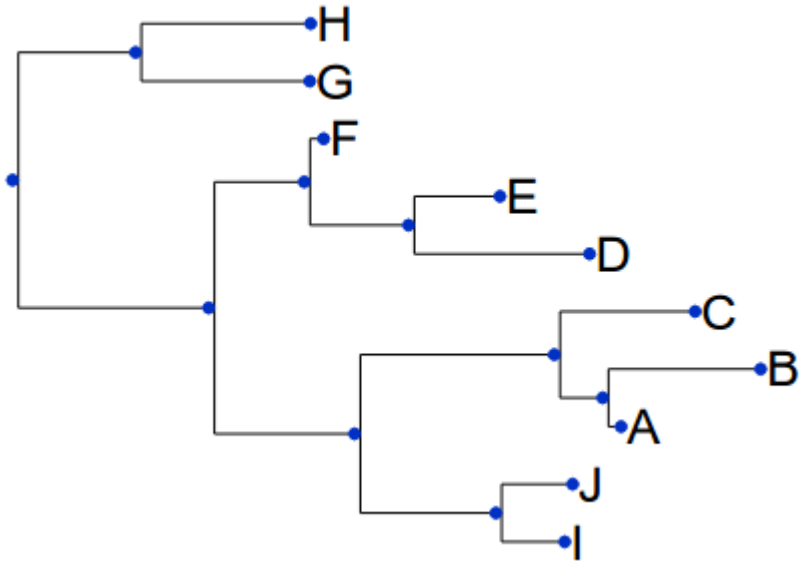
```
n [8]: t1 = ete3.Tree("example_tree_1.txt")

n [9]: t2 = ete3.Tree()

n [10]: t2.populate(10,["A","B","C","D","E","F","G","H","I","J"],random_branches=True)
```

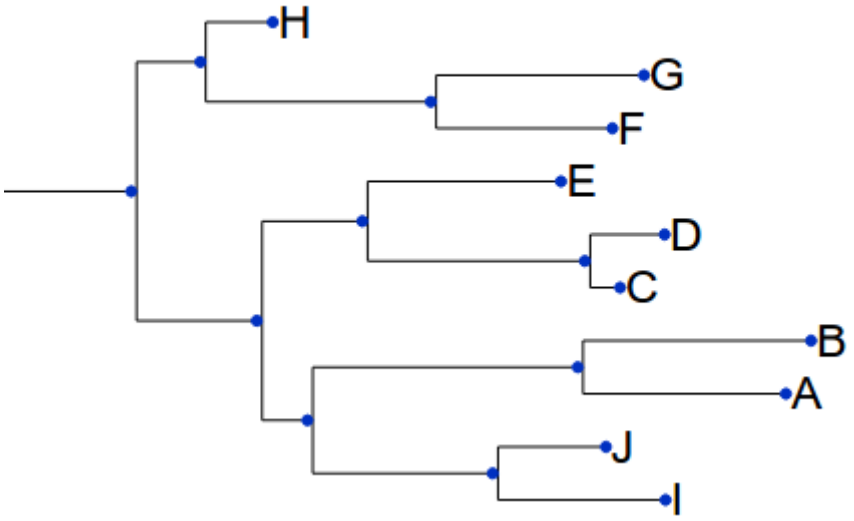
Now lets run the RF distance and print the results:

```
[n [12]: results = t1.robinson_foulds(t2)

[n [13]: for r in results:
   ....:     print r
```

The output of the command gives you:
The RF distance
The maximum RF distance
The list of common leaves
The partitions found in t1 not found in t2
The partitions found in t2 not found in t1
Discarded partitions in t1
Discarded partitions in t2

In this case there was a RF = 12, which means there were 12 partitions not shared between the two trees.



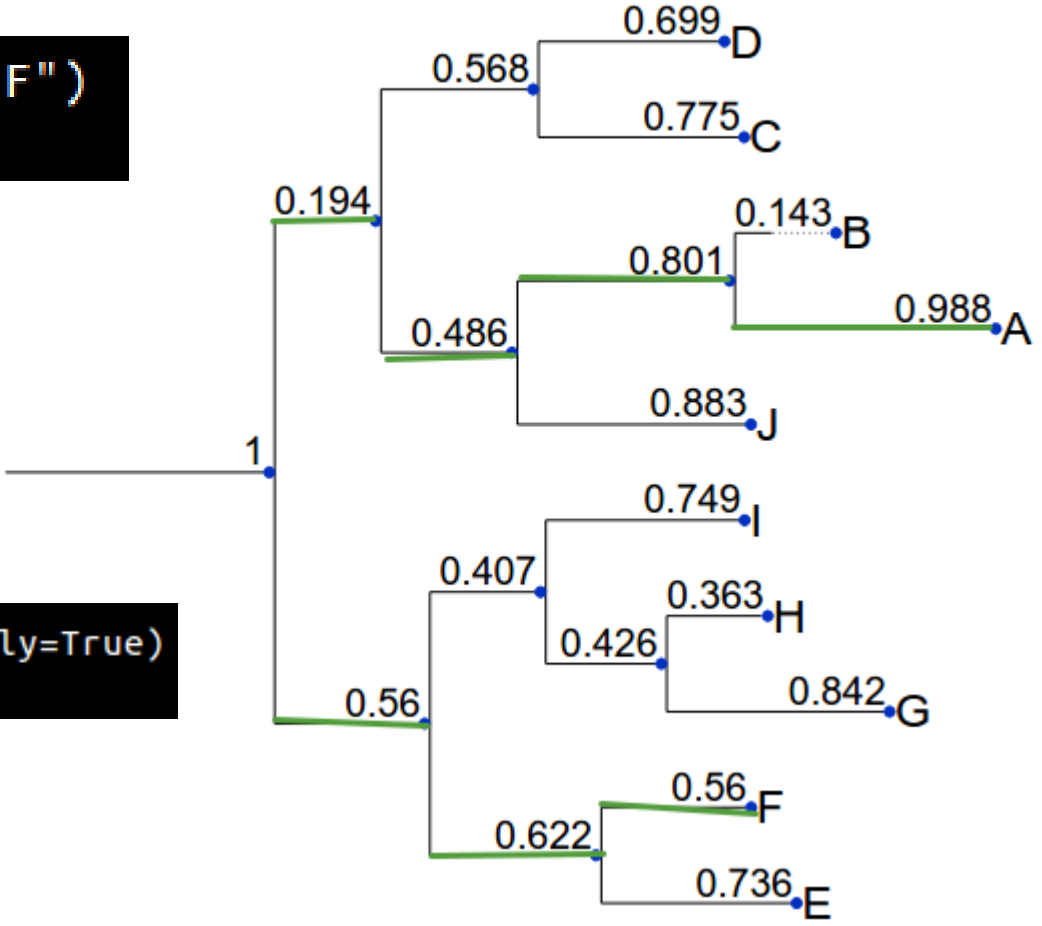




For trees with duplications you can try out the t.compare option. It will first divide the tree in all possible orthologous trees and then compare them all against all.

## Distances between branches in a tree.

Distances can be calculated based on branch lengths or on the number of nodes in the tree that you need to traverse to go from one node to another one.

```
n [18]: t.get_distance("A","F")
ut[18]: 4.210401892450086
```




```
n [20]: t.get_distance("A","F",topology_only=True)
ut[20]: 6.0
```

```
 [21]: t.get_distance("A")
t[21]: 2.4692736695678423
```

When only one leaf name is provided, it will calculate the distance from the leaf to the root.

## Modifying tree topologies

You could create a tree from scratch if you wanted to with the options add_child or add_sister.

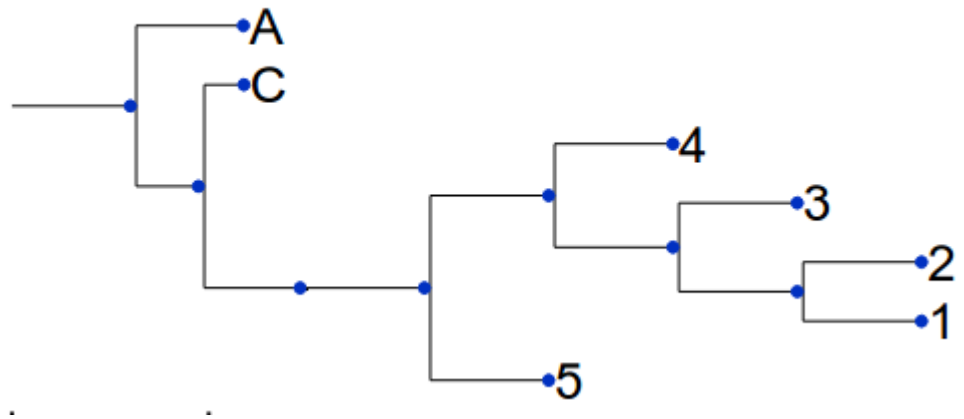
```
t = ete3.Tree()
```

```
ch1 = t.add_child(name="A",dist=0.9,support=70)
ch2 = t.add_child(name="B",dist=0.5,support=80)
ch3 = ch2.add_child(name="C",dist=0.3,support=10)
ch4 = ch2.add_child(name="D",dist=0.8,support=90)
```


You can do exactly the same from the graphical interface by right clicking on any node and using the add_children option.

With the same command you could also insert a populated node inside another one.

```
n [29]: t3 = ete3.Tree()

n [30]: t3.populate(5,["1","2","3","4","5"])

n [31]: ch5 = ch4.add_child(t3)
```



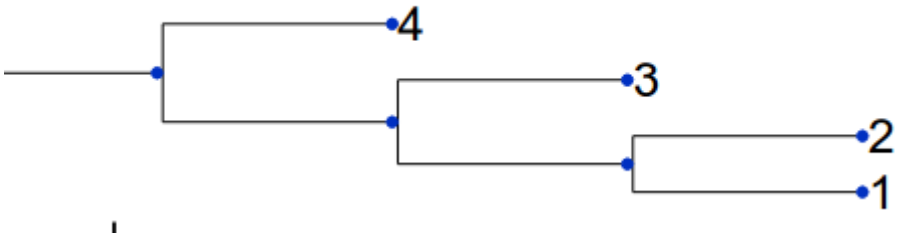
We can also remove nodes or partitions. There's two ways of doing this:

- Detach: you separate one subtree from the tree structure.

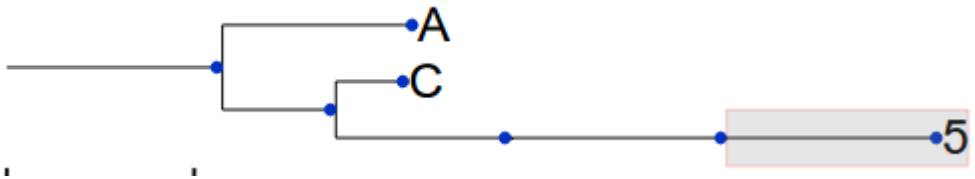
- Delete: you just remove the nodes but it will not affect its descendants (very usefull when you want to collapse nodes with less than 50% bootstrap support).

```
anc = t.get_common_ancestor("2","4")

subtree = anc.detach()
```

Detached subtree:

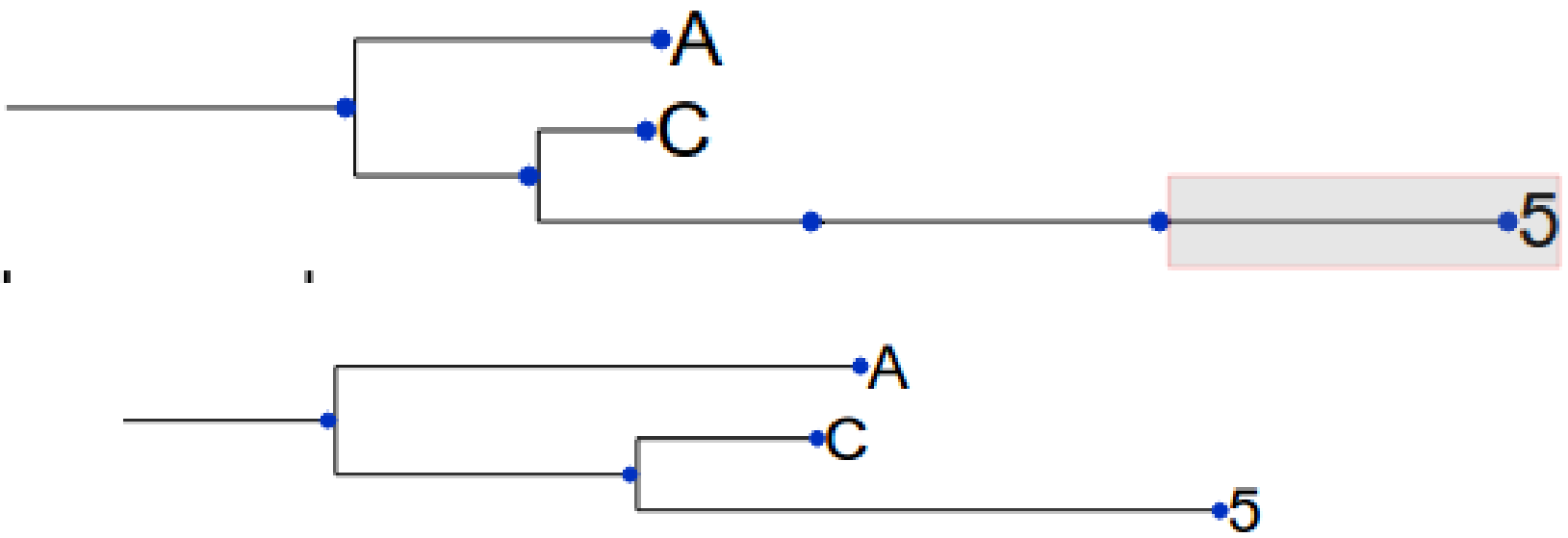



Remaining tree:



We have at least one node in the tree that only has one child, we probably want to remove this node but we don't want to delete its descendants:

```
n [39]: for node in t.traverse():
   ....:     if not node.is_leaf():
   ....:         if len(node.get_children()) == 1:
   ....:             node.delete()
   ....:
```




Note that the branch lengths of the deleted nodes are also deleted! If you want to keep them you will have to compute the total length and then modify the branch length of leaf 5.

# Prune trees

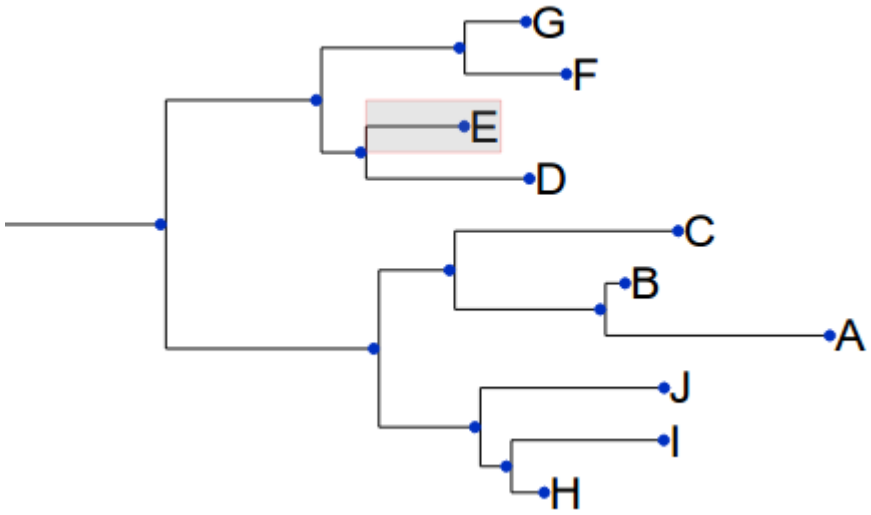
Given a tree, you may want to keep only certain leaves.

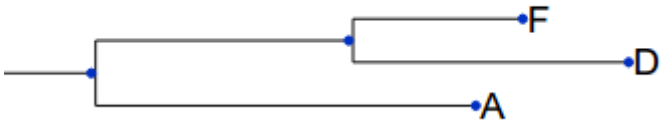
Lets start again from a random tree:

```
t = ete3.Tree()

t.populate(10,["A","B","C","D","E","F","G","H","I","J"],random_branches=True)
```

Now we will prune the tree so that we only keep leaves A,D and F.

```
[45]: t.prune(["A","D","F"])
```



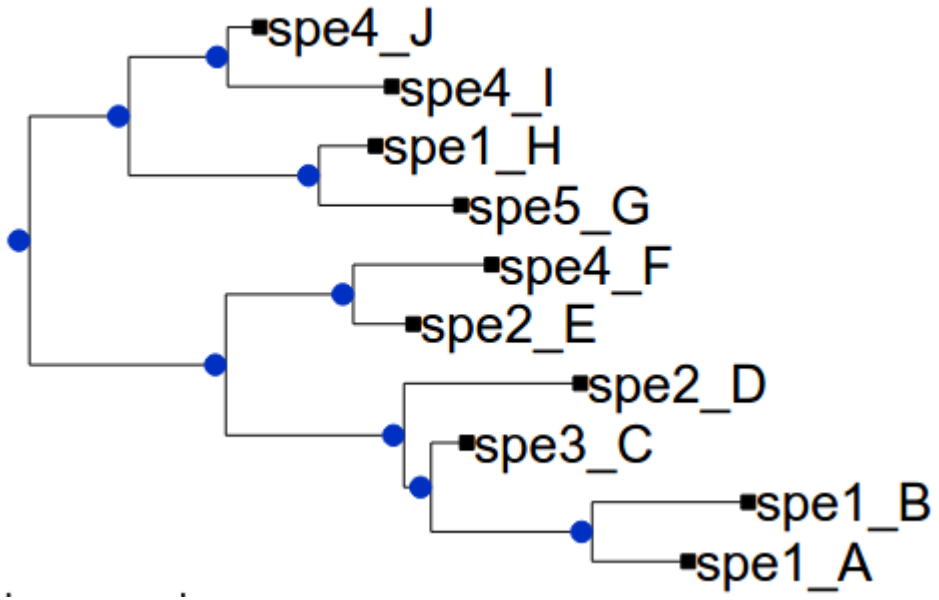

Note again that branch lengths are affected by this!

# PhyloTree

Until now we have worked with tree structures that could come from any given data. There is a special class in ETE that was developed to work specifically with phylogenetic trees. By default, a PhyloTree incorporates information about the species related to the leaf.

To load a PhyloTree:

```
In [51]: t = ete3.PhyloTree("example_tree_2.txt")
```




By default, ETE3 will assume that the first three letters of the leaf name show the species name. So for this example:

```
In [3]: t.get_species()
Out[3]: {'spe'}
```

To make sure the species name is loaded correctly you need to:

1.- Make sure all the leave names in your tree have the same format and that a part of it is the species tag.

2.- Create a function that will cut the leaf name and only return the species tag

```
In [4]: def get_species_name(node):
   ...:     return node.split("_")[0]
```

3.- Load the tree again with the newly created function

```
t = ete3.PhyloTree("example_tree_2.txt",sp_naming_function=get_species_name)
```

```
[6]: t.get_species()
[6]: {'spe1', 'spe2', 'spe3', 'spe4', 'spe5'}
```

# Rooting phylogenetic trees

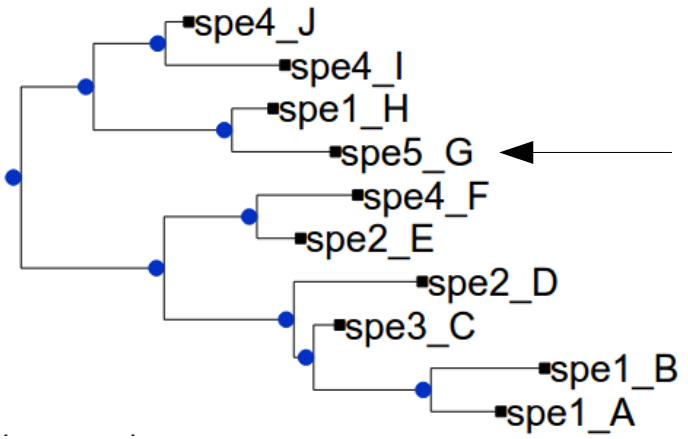
To properly interpret a phylogenetic trees, an usual requirement is that they are rooted. When working with one single tree, this can easily be done manually through the virtual interface. But to apply it on a large set of trees, we need automatic methods.

1.- How to know if a tree is rooted or unrooted: in ETE unrooted trees are represented with a trifurcation at the tree base, so you can know if a tree is rooted or unrooted just by counting the number of children

```
 [7]: len(t.get_children())
t[7]: 2
```

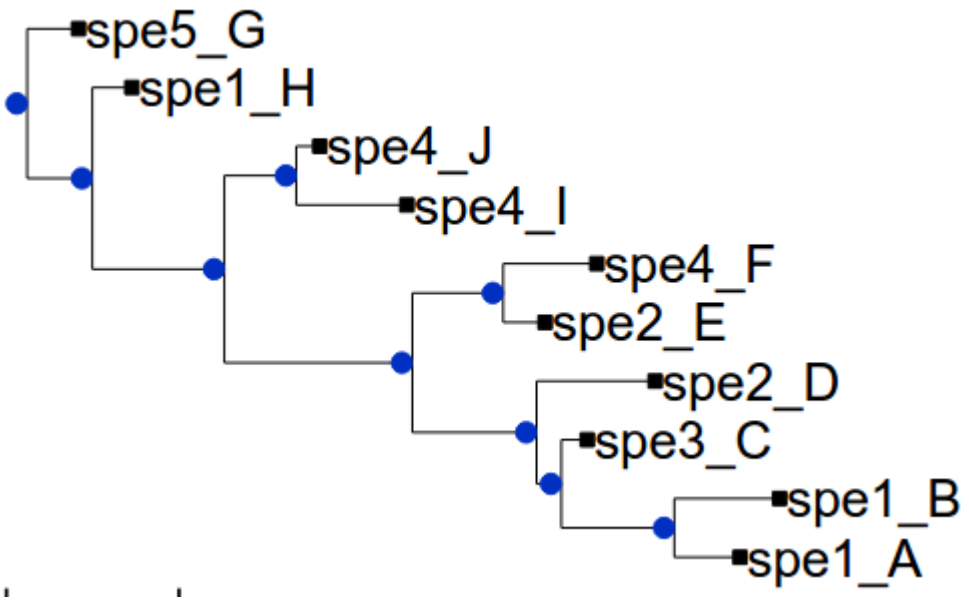
Even when a tree is rooted like this one, you may want to make sure the root is placed correctly because some programs return a rooted tree without knowing anything about where the root should really be placed.

To root the tree, the first thing you have to do is to choose the rooting place.



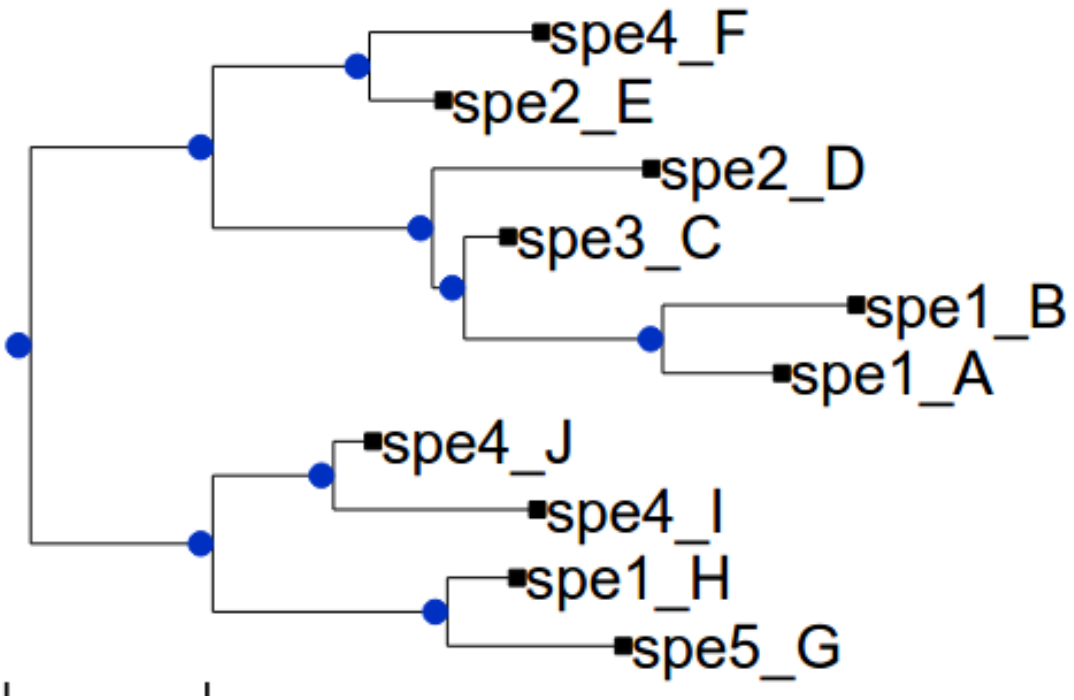
Root tree at a given leaf or node:

```
11]: root_point = t.get_leaves_by_name("spe5_G")[0]

12]: t.set_outgroup(root_point)
```



Root at midpoint: this is an automatic method that will search for the two most distant branches and place the root at the middle.

```
[14]: root_point = t.get_midpoint_outgroup()

[15]: t.set_outgroup(root_point)
```



Root by a preferred list of species.

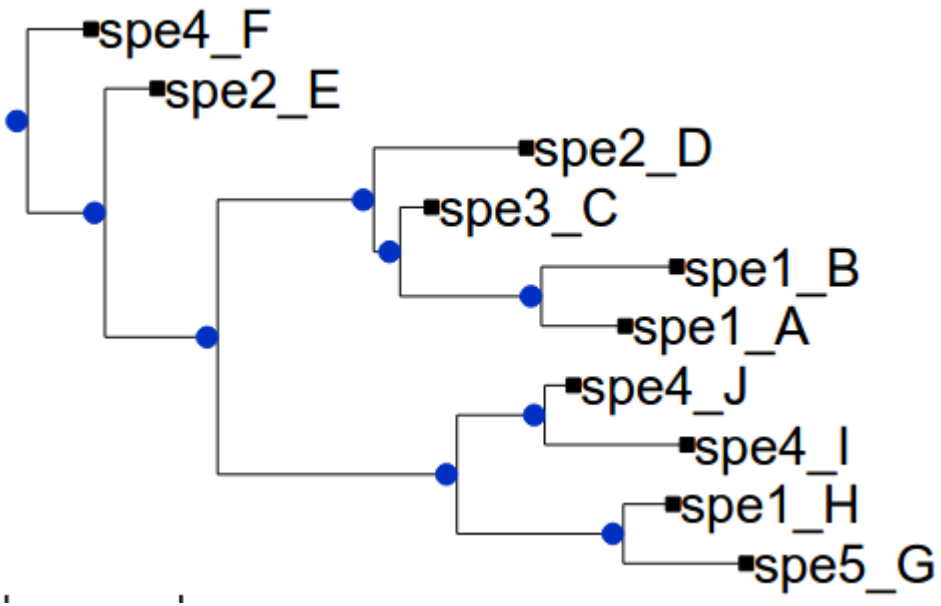
ETE3 allows you to define a python dictionary with a preferred order of species:

```
preference_dictionary = {"spe1":1,"spe2":2,"spe3":2,"spe4":4,"spe5":3}
```

The higher the number the more chance that the tree will be rooted there. So you should usually use 1 for your species of interest, and the highest numbers for your outgroups. More than one species can have the same number if you think that either option can serve as outgroup.

```
root_point = t.get_farthest_oldest_leaf(preference_dictionary)
```

```
t.set_outgroup(root_point)
```



## Detect evolutionary scenarios

Species overlap algorithm:

```
events = t.get_descendant_evol_events()
```

You can then iter through the events and obtain information about each event:

```
In [29]: print events[4].node

     /-spe4_J
  /-|
 |   \-spe4_I
-|
 |   /-spe1_H
  \-|
     \-spe5_G
```

```
In [30]: print events[4].etype
S
```

You can also obtain the evolutionary events for a single lineage (path from a species of interest to the root)

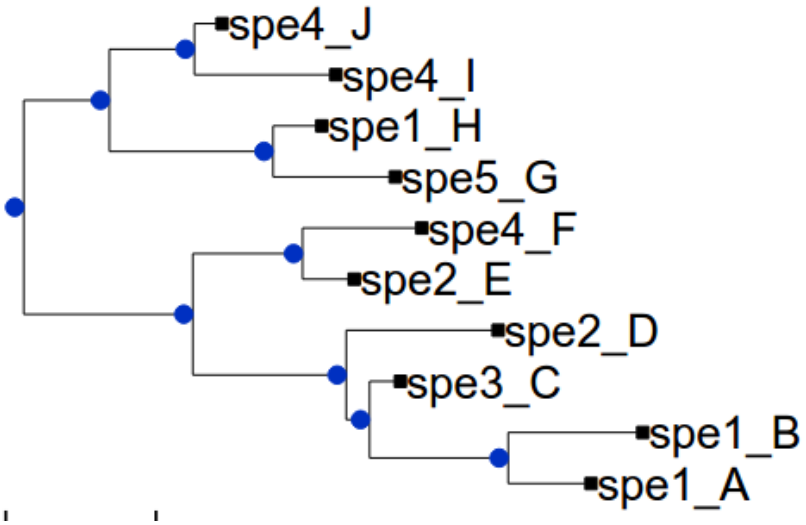
```
seed = t.get_leaves_by_name("spe4_I")[0]

events = seed.get_my_evol_events()
```

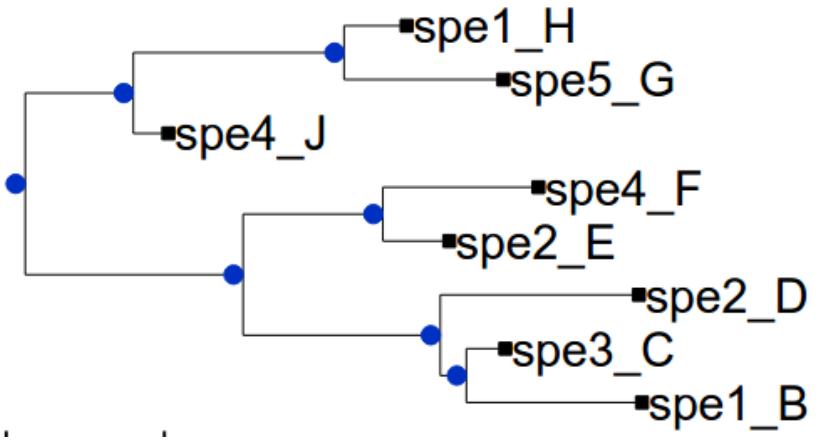
## Delete species specific duplications

To build a concatenated tree the chosen genes need to be in single copy in each of the species. In some cases, that may be complicated (i.e. we have a hybrid). One way to increase the gene sampling is to check out the gene trees and delete species specific duplications and select one of the copies at random. ETE3 implements a function that can do that:

```
t = ete3.PhyloTree("example_tree_2.txt",sp_naming_function=get_species_name)
```

```
collaps_tree = t.collapse_lineage_specific_expansions()
```



## Tree visualization

How to visualize a tree and print it to a file:

```
t = ete3.Tree("example_tree_1.txt")

t.show()
```

This command will simply open the visual interface. From the interface you can print the image into a PDF file by pressing on the PDF icon.

You can print the image in different formats by using the render function:

```
t.render("image.png")
```

```
t.render("image.svg")
```

```
t.render("image.pdf")
```

Just by changing the extension of the output file, you tell ETE in which format you want to print your image.

# Tree style

To change the visualization format of a tree, you first need to create a tree style, that will be applied to the tree:

```
ts = ete3.TreeStyle()
```

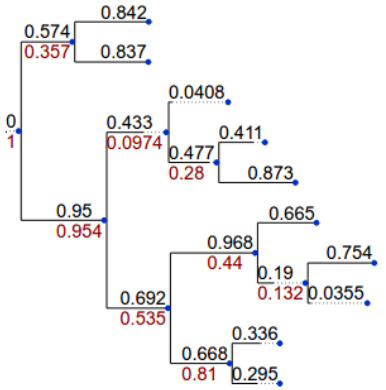
The tree style controls the main visualization features of the tree.

```
[10]: ts.show_branch_length=True

[11]: ts.show_branch_support=True

[12]: ts.show_leaf_name=False
```

We can for instance ask for the branch lengths and supports to be shown and tell the tree not to show the tree names.

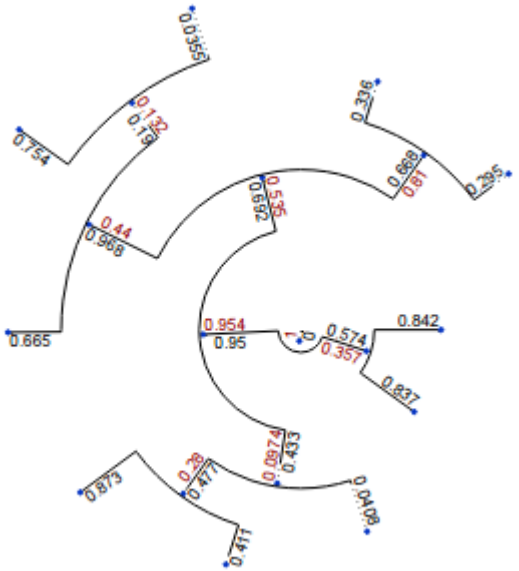
To apply the style you just created, you need to use a tree_style=ts either in .show() or in .render()

```
t.show(tree_style=ts)
```



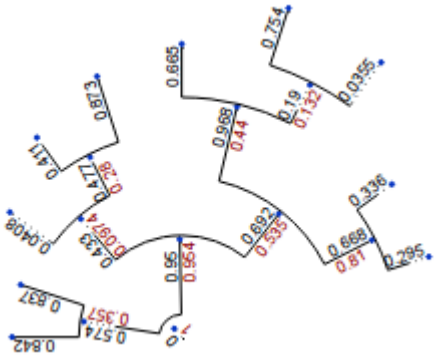
# Tree style

Other things the tree style can control:



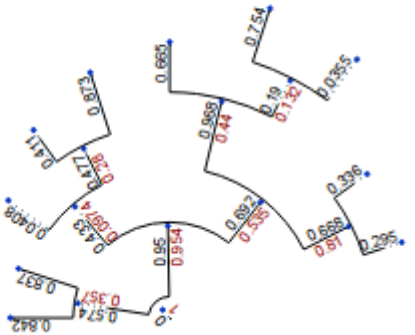
```
ts.mode="c"
```

+



```
ts.arc_span = 180

ts.arc_start = -180
```

Example



+

```
ts.title.add_face(ete3.TextFace("Example",fsize=20),column=0)
```

# Node faces

Faces can be used to add additional information to the trees. Their position in relation to the tree can be found in this graph:


To add faces to a tree, you first need to create a Face object and then you need to add it to one or several nodes in the tree (can be internal nodes or leaves).

Here are the main kinds of faces (this assumes you have exported the Faces module from ete3: from ete3 export Faces):

```
N = AttrFace("name", fsize=30)
(faces.TextFace("Drawing your own Qt Faces", fsize=15)
chimpFace = faces.ImgFace(img_path+"chimp.png")
C = CircleFace(radius=node.weight, color="RoyalBlue", style="sphere")
seqFace = SeqMotifFace(seq, gapcolor="red")
```

Each face has a set of parameters that can be modified to adapt the image to what you would like to draw.

*class* **TextFace**(*text, ftype='Verdana', fsize=10, fgcolor='black', penwidth=0, fstyle='normal', tight_text=False, bold=False*)

Static text Face object

**Parameters:**

- **text** – Text to be drawn
- **ftype** – Font type, e.g. Arial, Verdana, Courier
- **fsize** – Font size, e.g. 10,12,6, (default=10)
- **fgcolor** – Foreground font color. RGB code or color name in `SVG_COLORS`
- **penwidth** – Penwdith used to draw the text.
- **fstyle** – "normal" or "italic"

*class* **ImgFace**(*img_file, width=None, height=None, is_url=False*)

Creates a node Face using an external image file.

**Parameters:**

- **img_file** – path to the image file.
- **width** (*None*) – if provided, image will be scaled to this width (in pixels)
- **height** (*None*) – if provided, image will be scaled to this height (in pixels)
- **is_url** (*False*) – if True, img_file is considered a URL and the image is automatically downloaded

## Node faces: how to place them

Once you have created a face, you will have to add it to the tree. To do that you first need to select a node where the face needs to be placed. And then decide where you will want to place it.


Here is how we would place a node in the first column and asked it to be aligned

```
faces.add_face_to_node(descFace, node, column=0, aligned=True)
```

Another way to add a face to the tree, in this case it will be placed above and below a branch.

```
t.add_face(hola, column=0, position = "branch-top")
t.add_face(mundo, column=1, position = "branch-bottom")
```

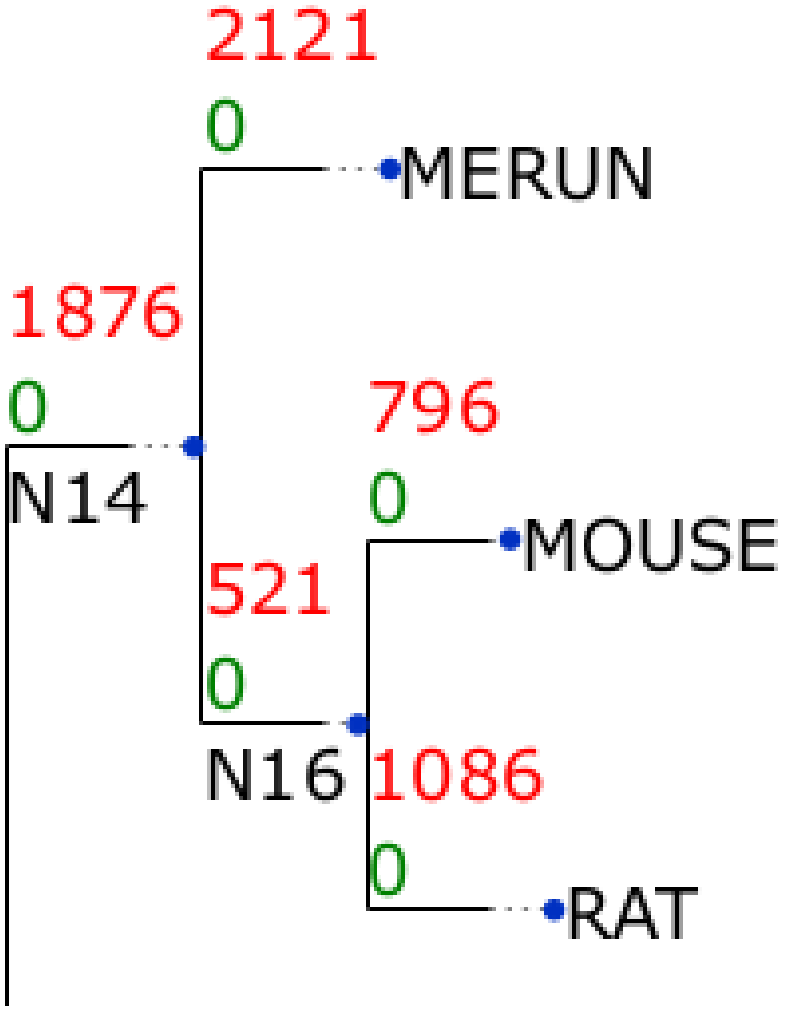
## Layouts

The layouts are functions that allow you to make modifications on each node before they have been drawn. These functions accept as input a given node and allow you to decide what to do with that node.

Example layout:

```
def mapped_data_layout(node):
        if "loss" in node.features:
                F = ete3.AttrFace("loss",fgcolor="red")
                ete3.add_face_to_node(F,node,column=0,position="branch-top")
        if "gain" in node.features:
                F = ete3.AttrFace("gain",fgcolor="green")
                ete3.add_face_to_node(F,node,column=0,position="branch-top")
        if "nodeName" in node.features and not node.is_leaf():
                F = ete3.AttrFace("nodeName")
                ete3.add_face_to_node(F,node,column=0,position="branch-bottom")
        if node.is_leaf():
                N = ete3.AttrFace("name")
                ete3.add_face_to_node(N,node,column=0)
```

This layout prints the number of gained genes in a lineage on top of a branch in green, the loss of genes in a lineage also on top of the branch but in red, the internal node on the bottom of the branch, and finally the node name.

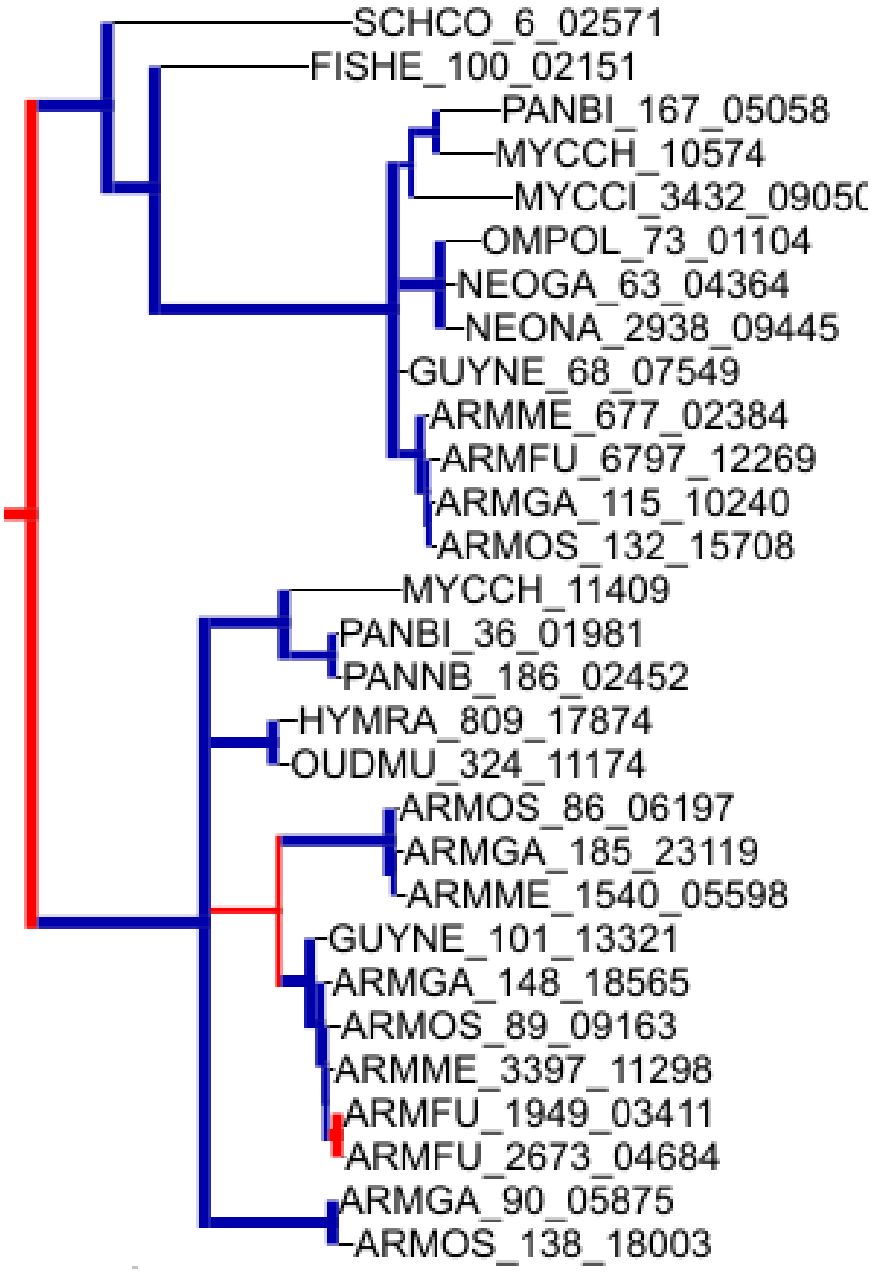

2121
0
MERUN
1876
0
N14
796
0
MOUSE
521
0
N16
1086
0
RAT

Another example:
1.- Delete all node with support < 50 (create polytomies)
2.- Create several node styles, each slightly different in terms of colour and thickness, so that it can be assigned to branches with different supports.

```
nstyle90S = ete3.NodeStyle()
nstyle90S["size"] = 0
nstyle90S["vt_line_width"] = 4
nstyle90S["hz_line_width"] = 4
nstyle90S["vt_line_type"] = 0 # 0 solid, 1 dashed, 2 dotted
nstyle90S["hz_line_type"] = 0
nstyle90S["vt_line_color"] = "#0000aa"
nstyle90S["hz_line_color"] = "#0000aa"
```

```
def bootstrap_layout(node):
        node.img_style["size"] = 0
        if node.support >= 90.0:
                if node.evoltype == "S":
                        node.set_style(nstyle90S)
                elif node.evoltype == "D":
                        node.set_style(nstyle90D)
        elif node.support >= 75.0:
                if node.evoltype == "S":
                        node.set_style(nstyle75S)
                elif node.evoltype == "D":
                        node.set_style(nstyle75D)
        elif node.support >= 50.0:
                if node.evoltype == "S":
                        node.set_style(nstyle50S)
                elif node.evoltype == "D":
                        node.set_style(nstyle50D)
```

The layout will be the one in charge to assign the different node styles to each node according to the support and the evolutionary support.


SCHCO_6_02571
FISHE_100_02151
PANBI_167_05058
MYCCH_10574
MYCCI_3432_0905(
OMPOL_73_01104
NEOGA_63_04364
NEONA_2938_09445
GUYNE_68_07549
ARMME_677_02384
ARMFU_6797_12269
ARMGA_115_10240
ARMOS_132_15708
MYCCH_11409
PANBI_36_01981
PANNB_186_02452
HYMRA_809_17874
OUDMU_324_11174
ARMOS_86_06197
ARMGA_185_23119
ARMME_1540_05598
GUYNE_101_13321
ARMGA_148_18565
ARMOS_89_09163
ARMME_3397_11298
ARMFU_1949_03411
ARMFU_2673_04684
ARMGA_90_05875
ARMOS_138_18003

Once the layout has been created, it needs to be assigned to the TreeStyle so that it will be used when drawing the tree. This is as simple as:

```
ts = ete3.TreeStyle()
ts.layout_fn = bootstrap_layout
ts.show_branch_support = True
```

Some tips:

1.- In some cases, if you want to change the leaf names in the layout, you will have to ask the TreeStyle to omit showing leaf names. This is because they are included as a new face by the layout.

2.- If, after assigning your layout to the TreeStyle, you change something on the function, you will have to re-assign it again, else it will not detect those modifications.