

Chapter 2

Phylogenetic Data in R

2.1 Objectives

The objectives of this chapter are to introduce the user to how phylogenetic information is stored, presented, and manipulated in R. We will cover primarily the class “phylo” in this chapter since that is the class that is most frequently utilized in phylogenetic diversity and comparative analyses in R. By the end of this chapter the user should have a basic command of how to plot phylogenies and extract information from the data files. The chapter is designed for beginners, and users familiar with using phylogenies in R may quickly scan this chapter to refresh.

2.2 Loading Phylogenies into R and the Structure of the “Phylo” Class

This section deals with loading phylogenetic data files into R as a “phylo” class and how the data are structured inside R. The focus will be primarily on learning the basics nuts and bolts of phylogenetic data in R. As such there will be a number of aspects of phylogenetic data that are not germane to the goals of this book that will not be covered. We will begin by simply reading in an example phylogeny that is in the parenthetical Newick format. This can be accomplished using the `read.tree()` function in the *ape* R package. Thus, we must first load the *ape* package and then use the function.

```
> library(ape)
> my.phylo <- read.tree("my.phylo.newick.file.txt")
```

The online version of this chapter (doi: [10.1007/978-1-4614-9542-0_2](https://doi.org/10.1007/978-1-4614-9542-0_2)) contains supplementary material, which is available to authorized users

To print the original file to your R console so that you can see the original Newick format simply use the `write.tree()` function.

```
> write.tree(my.phylo)
```

Alternatively, you could also open the original file in a text editor. We can also read a Nexus formatted phylogeny into R using the `read.nexus()` function in the *ape* package.

```
> my.nexus.phylo <-  
read.nexus("my.phylo.nexus.file.txt")
```

To print the original Nexus format to your R console so that you can see the original Nexus format simply use the `write.nexus()` function.

```
> write.nexus(my.phylo)
```

The Nexus file could also be opened with a text editor or with the program Mesquite [20]. Both files are now stored as class “phylo” objects in R. Thus, the two files that were once in different formats are now stored using a structure in R allowing us to use either version in the following analyses in this chapter and the majority of this book. While Nexus files can and often do hold much more information along with the phylogenetic tree such as trait data and sequence alignments, in this instance the Nexus file does not hold additional information and we will use the Newick version stored in the `my.phylo` object for the remaining analyses.

Now that our phylogeny is in R we should explore its contents and structure first before we do any analyses just like you should do for any other data file you would have in R. To do this we can first simply type the name of the object.

```
> my.phylo
```

We see that our example phylogeny contains 26 tips and 25 internal nodes. The phylogeny also has “terminal nodes,” which are the tips. In some cases a phylogenetic tree may not be fully bifurcating. That is, a single branch may not split into two branches at an internal node and may branch simultaneously into three or more branches. This is called a “polytomy.” A polytomy can be further classified as a “hard polytomy” where the branching of three or more lineages from one is to signify an actual simultaneous divergence or a “soft polytomy” where the branch of three or more lineages from one is used to signify ignorance regarding who is most closely related to whom between the derived and ancestral lineages. Typically, the polytomies that you will encounter will be soft polytomies. A quick way to tell whether your phylogeny probably contains a polytomy is that the number of internal nodes is less than the number of tips minus 1.

The next item printed out by R is a series of tip labels for the first six tips. The next item lists node labels if they exist for our phylogeny. In many cases you will not have node labels in your phylogeny. Finally, we see that the phylogeny is rooted and has branch lengths. This is all useful information that we can quickly glean, but

it is simply printed on our screen and is not useful for more detailed investigations into our phylogeny. In some cases we can simply ask R whether our phylogeny has a particular structure. For example, we can ask if the phylogeny is rooted or if it is ultrametric (i.e., all tips end at the same point as is true for a phylogeny of extant species scaled to time).

```
> is.rooted(my.phylo)

> is.ultrametric(my.phylo)
```

Though, even the answers to these questions are not only a fraction of what we might want to know or extract from our phylogenetic tree in R and we therefore need to learn more about how the phylogenetic data is structured in R. To learn more about how our phylogenetic data is structured, we can ask for the names of our phylogeny object.

```
> names(my.phylo)
```

We see that five names: “edge,” “tip.label,” “Nnode,” “node.label,” and “edge.length” are reported. We can extract each of these from the object using the \$ symbol and the name. For example, to examine what the “edge” is we can do the following:

```
> my.phylo$edge
```

This results in a matrix with two columns, and the number of rows corresponds to the number of branches in our phylogeny. The values in the first column correspond to the internal node from where the branch originates, and the values in the second column correspond to the internal or terminal node where the branch ends. The next name we can examine is “tip.label.”

```
> my.phylo$tip.label
```

This returned a vector containing all of the names of the taxa on the tips of our phylogeny. We will examine this later, but the order of the names in this vector is from the species on the bottom of the phylogeny when plotted using `plot(my.phylo)`, and the last name is the species on the top of the phylogeny when plotted using `plot(my.phylo)`. We now examine what our phylogeny object holds under the name “Nnode.”

```
> my.phylo$Nnode
```

The number reported is simply the number of internal nodes in our phylogeny. Next we can ask what is contained under the “node.label” name.

```
> my.phylo$node.label
```

This produces a vector of the internal node labels in our phylogeny with one label per node. In this instance the node labels are simply the numbers 1 through 25. Recall, that many phylogenies will not have node labels or may have labels for only some of the internal nodes. Lastly, we can ask what is under the “edge.length” name in our object.

```
> my.phylo$edge.length
```

A vector of branch (i.e., edge) lengths is returned. The length of this vector is equal to the number of branches in our phylogeny, and the order of the values corresponds to the order of the branches described under `my.phylo$edge`. Thus, if we wanted to make a matrix that had the information regarding the node numbers for the beginning and end points of each branch in the first two columns and the length of that branch in the third column we could do the following:

```
> cbind(my.phylo$edge, my.phylo$edge.length)
```

We now have investigated the basic structure of our phylogenetic data object in R. In Sect. 2.4 we will discuss how to modify this data or to use it to calculate additional information, but first we should discuss how to plot phylogenetic trees in R because it will help us visualize downstream manipulations we will perform.

2.3 Plotting Phylogenetic Trees in R

In any situation it is always a good idea to first look at your data once it is read into R. In the case of a table or matrix one can plot the data as a histogram or just look at the numbers themselves. In the case of phylogenetic trees, a first approach can be to examine the object using the approach we used in the previous section. After this initial examination it is good to simply plot the phylogenetic tree to further investigate the data to assure there is nothing out of place or “odd” about your data. There are two easy ways to plot a phylogenetic tree in R. The first just uses the generic `plot()` function (Fig. 2.1):

```
> plot(my.phylo)
```

If we would like to decrease the size of the names of terminal taxa, in this case species, in the plot we can adjust their size by changing the `cex` value to smaller than the default of one. We can also add a scale bar using a second function called `add.scale.bar()` and providing the length desired for the scale bar (Fig. 2.2).

```
> plot(my.phylo, cex = 0.8)
> add.scale.bar( , length = 0.1)
```

The `plot()` function is useful for a quick examination, but it can be limited as it is not specialized for phylogenetic data. The function `plot.phylo()` in the *ape*

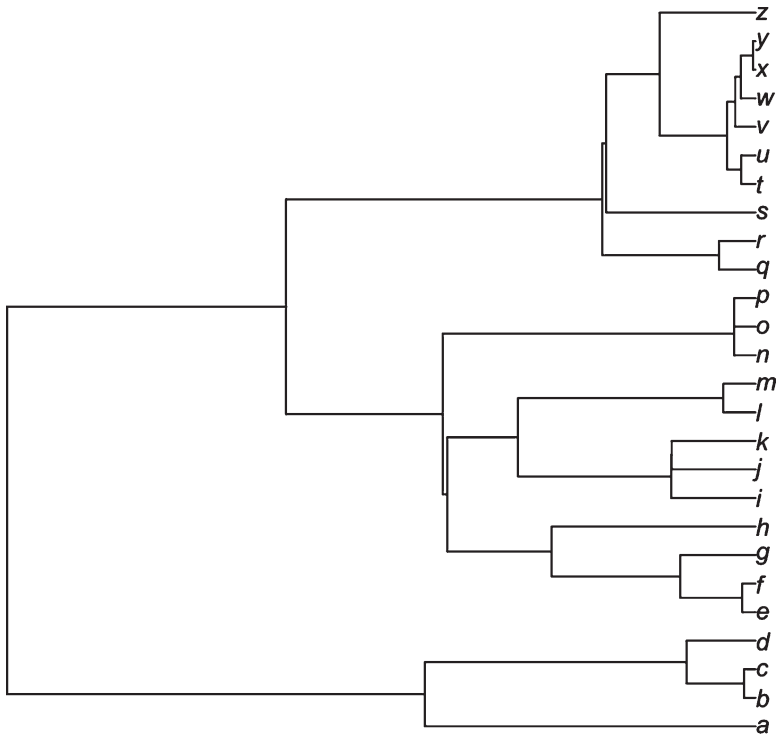


Fig. 2.1 The phylogenetic tree for our dataset

package is more flexible and designed to specifically plot phylogenetic objects. This function allows the user to display the phylogeny in different styles or types and to manipulate species names and tree branches. For example, we can plot our phylogeny as a “fan,” as is often done for phylogenies much larger than ours, with gray branches (Fig. 2.3).

```
> plot.phylo(my.phylo, type = "fan", show.tip.label = TRUE,
show.node.label = FALSE, edge.color = "gray", edge.width = 1,
tip.color = "black")
```

As you can see from the code there are many options available to us regarding how to plot the phylogeny including whether or not to show labels, how to color the branches, and how to color the terminal taxa. This list does not include all of the possible ways to alter the appearance of the phylogeny using `plot.phylo()`. For a full description of all the possibilities, simply type `?plot.phylo`.

Simply plotting your phylogeny in this way or with some other exotic format is fun and can be useful and you may find that you may want to add additional information to “decorate” your tree and this may not be possible using the `plot.phylo()` function. For example, in the previous section we looked at the node numbers for each branch, but we may not know how those node numbers are arrayed

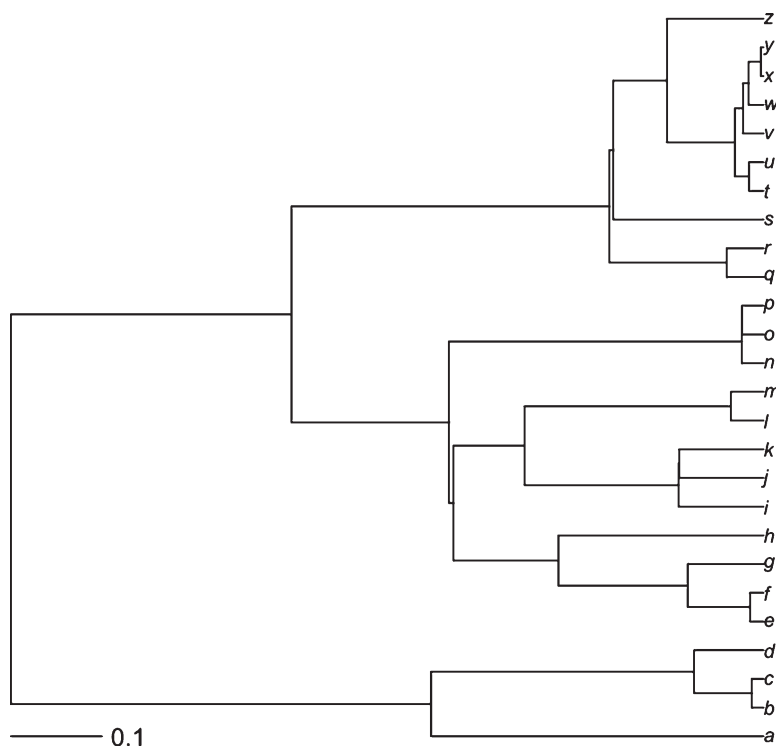


Fig. 2.2 The phylogenetic tree for our dataset with smaller labels and a branch length scale bar

on our phylogeny. To solve this problem we could first plot our phylogeny, in this example with `plot()`.

```
> plot(my.phylo)
```

Next we could simply ask R to place the node number in black for each internal node in a gray box on that node using the `nodelabels()` function (Fig. 2.4).

```
> nodelabels( , col = "black", bg = "gray")
```

You will note that the node label nearest to the root of the phylogeny is one more than the number of species in our phylogeny. That is because the first 26 node numbers correspond to the 26 species in our phylogeny. To see this you can use the `tiplabels()` function (Fig. 2.5).

```
> tiplabels( , col = "black", bg = "gray")
```

A similar procedure could be used if you had a value for each internal node sorted in the same order as the node numbers. For example, you may have an ancestral trait value estimated for each internal node that you would like to assign. This could be

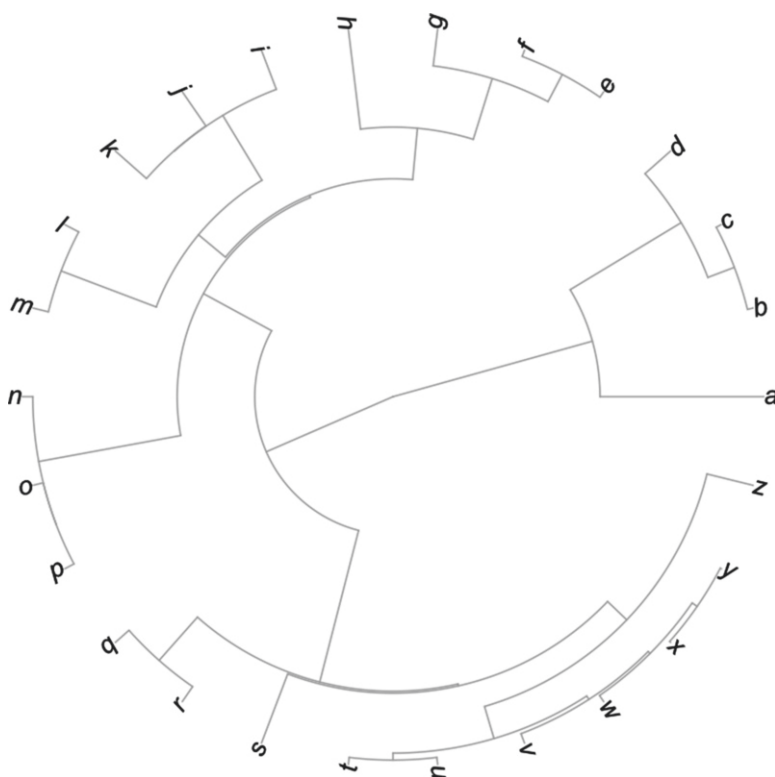


Fig. 2.3 The phylogenetic tree for our dataset displayed as a circular phylogeny or “fan” type phylogeny

done simply by providing a vector of the values to the `node.labels()` function. There are a number of additional ways to display and decorate your tree with differing degrees of usefulness. It is not the goal to cover those more exotic approaches presently, and I will leave you to explore the wonderful diversity of ways one can plot a phylogeny in R. For the present time we can be satisfied with simply visualizing the basic structure of our phylogeny and we can proceed with how to manipulate that information or to calculate additional information from the basic structure.

2.4 Manipulating and Calculating Additional Information from Phylogenetic Trees in R

We now know how to examine the structure of our phylogenetic data objects in R and how to plot the phylogenies for visualization. The next step is to learn how to manipulate or extract additional information from the phylogenetic trees.

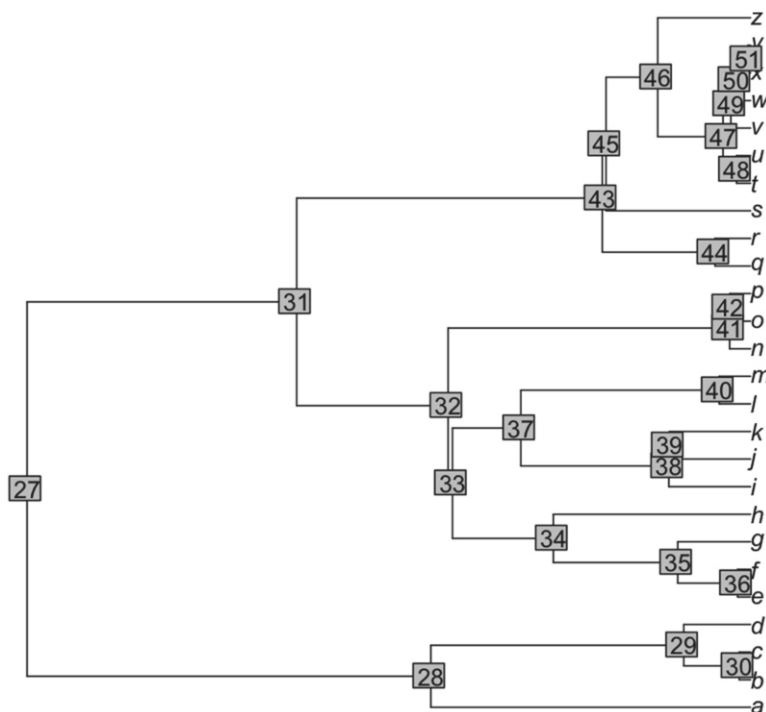


Fig. 2.4 The phylogenetic tree for our dataset with internal nodes labeled with their node numbers

We will begin by learning how to extract subsets of our original phylogenetic tree. This can be accomplished in two ways—by extracting entire clades or by pruning particular taxa out of the phylogeny. Both approaches can be useful for an ecologist. The first is useful if one wanted to perform an analysis only on a particular clade and the second is useful if one is given a phylogeny that contains more species than are in the community or trait data set being analyzed. To extract individual clades from our phylogeny, we can use the `subtrees()` function. This function takes an input phylogeny and produces a list where each element is a phylogeny object of class “phylo” containing the species derived from an individual internal node in the phylogeny. Thus, the length of the list should be equal to the number of the internal nodes in the phylogeny.

```
> my.subtrees <- subtrees(my.phylo)
```

We can look at the 15th individual subtree by asking for the 15th element in the list produced by the `subtrees()` function.

```
> my.subtrees[[15]]
```

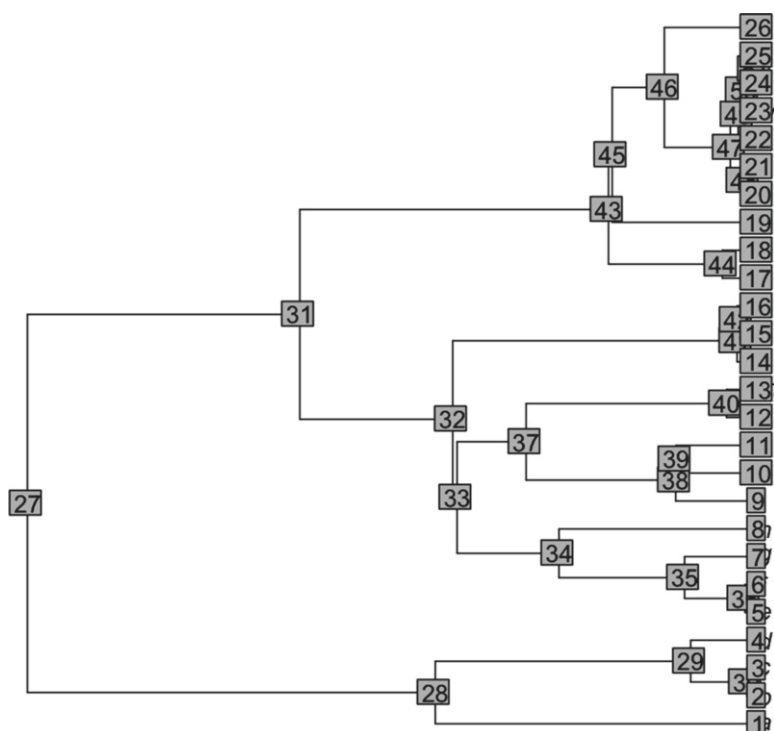



Fig. 2.5 The phylogenetic tree for our dataset with internal and terminal nodes labeled with their node numbers

We see that this particular subtree contains three taxa and two internal nodes. We can visualize this subtree by plotting it (Fig. 2.6).

```
> plot(my.subtrees[[15]])
```

The alternative to extracting all subtrees or clades from the original phylogeny is to selectively prune individual taxa out of the phylogeny. This can be done using the `drop.tip()` function, which takes a phylogeny object and a vector of names to be pruned from the phylogeny (Fig. 2.7).

```
> drop.tree <- drop.tip(my.phylo, c("e", "j", "s"))
> plot(drop.tree)
```

We can see that the three species we specified were pruned out of our original phylogeny. This approach can therefore be a powerful tool, but the downside is that you must know all of the names you don't want to keep whereas it may be easier to know all of the names you do want to keep. Later in the book we will address this situation.

Fig. 2.6 A plot of the 15th subtree or clade from our example phylogeny

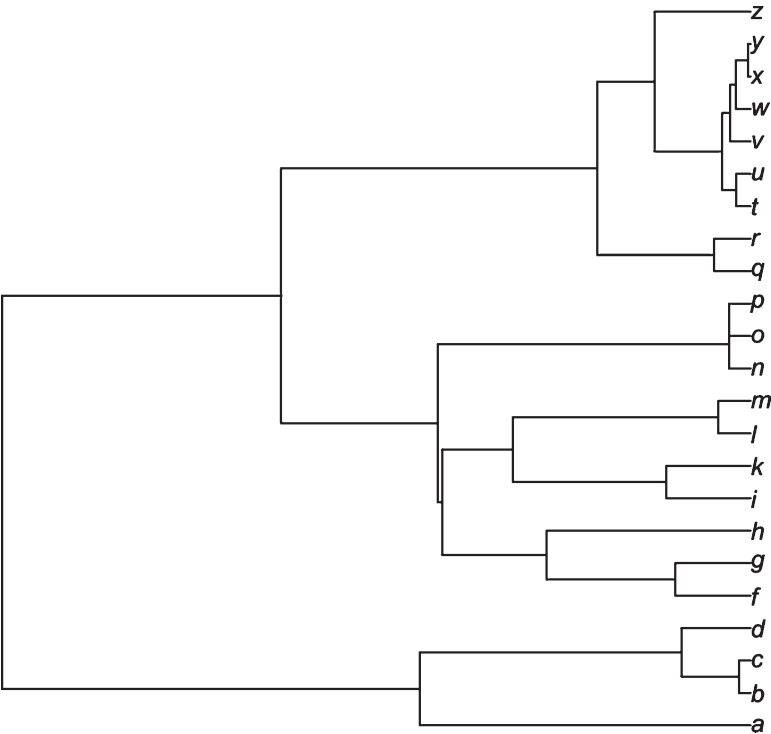
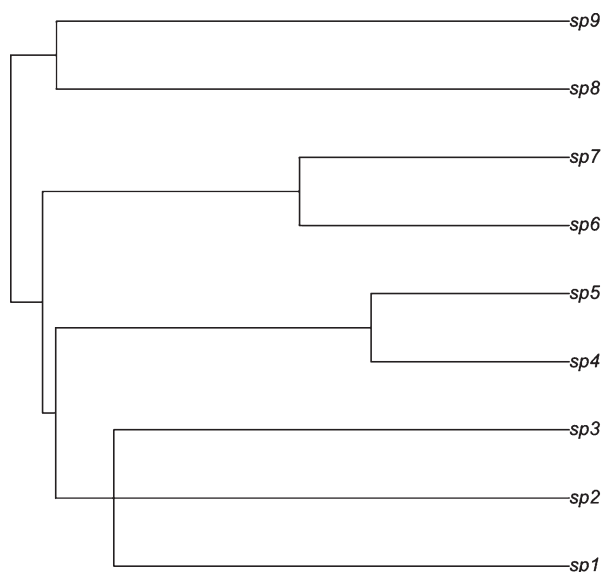


Fig. 2.7 A plot of our example phylogeny with species “e,” “j,” and “s” removed or pruned. Compare with Fig. 2.2 to visualize the difference

Fig. 2.8 A plot of our example phylogeny containing a single soft polytomy indicating uncertainty regarding the relatedness of sp1, sp2, and sp3

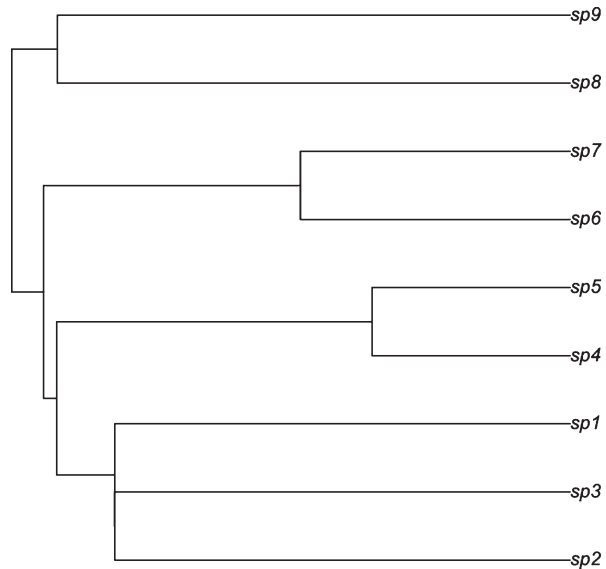


Next let us consider a situation where our phylogeny is not fully bifurcating. In many cases such a phylogeny will cause no problems for the analyses we will cover in this book, but in some cases the functions check that the phylogeny is fully bifurcating before running the analysis. If the phylogeny is not bifurcating the function will not run. Because our original example phylogeny was fully bifurcating, let us read in an example phylogeny with a single node in it that splits into three daughter lineages and not two (Fig. 2.8).

```
> my.poly.phylo <- read.tree("example.poly.txt")
> my.poly.phylo
> plot(my.poly.phylo)
```

The node with three branches emerging from a single node represents our uncertainty regarding the true relationship between these three species. There are three possibilities: the first two lineages are more closely related to each other than the third lineage, the first and third lineages are more closely related to each other than the second lineage, or the second and third lineages are more closely related to each other than the first lineage. When this uncertainty occurs it is often good practice to randomly resolve the polytomy several times and rerun the analysis each time to estimate the sensitivity of the result to the uncertainty (e.g., [21]). In the above simple case there are only three possibilities, but imagine a situation with multiple polytomies some of which are nested. The number of possibilities could be quite large. Further, when a polytomy of three lineages is resolved, a new node must be added to the phylogeny and the distance from the original polytomous node to the new node is unknown and varying this distance introduces a massive number of

Fig. 2.9 A plot of our example polytomous phylogeny where we have randomly resolved the polytomy for sp1, sp2, and sp3. Note that the phylogenetic tree still appears to contain a polytomy



possibilities to consider. A first step that some phylogenetic programs, such as Mesquite [20], take is simply placing the new node at half the branch length from the polytomous node to its daughter node. Unfortunately, as far as I am aware, functions currently in R do not have this particular capability, but there is a function called `multi2di()` that randomly resolves polytomies in your phylogeny, but as we will see in a second the new branch lengths separating the species are zero. In this sense the `multi2di()` function tricks the R function you are trying to run into thinking the phylogeny is bifurcating, but in reality it has placed a new node at zero distance from the original polytomous node. Despite this, the `multi2di()` function can be a quick and unbiased way to manipulate your polytomous phylogeny so that the function you are interested in implementing will run.

```
> my.di.phylo <- multi2di(my.poly.phylo)
> my.di.phylo
```

We see that the new phylogeny has one more node than the original phylogeny we read into R and that the number of internal nodes is now one less than the number of tips. Though when we plot the bifurcating phylogeny we see that the tree still looks to have a polytomy. This is because the new node randomly resolving the polytomy has been placed zero distance for the original polytomous node (Fig. 2.9).

```
> plot(my.di.phylo)
```

Our next goal is to extract the branching time for each internal node for an ultrametric phylogeny. This can be done simply using the `branching.times()` function in the *ape* package.

```
> branching.times(my.phylo)
```

The output is a vector of values with a length equal to the number of internal nodes in the phylogeny. The order of the nodes is from the root of the phylogeny towards the tips. It is important to know that this function only works for an ultrametric phylogeny. If the phylogeny is not ultrametric some nodes may have a negative branching time because they are distal of some terminal taxa.

The last piece of information that we can extract from a phylogeny that we will discuss in this chapter is a matrix depicting the phylogenetic distance between each pair of terminal taxa or a matrix that depicts the amount of shared branch length between each pair of terminal taxa. The first type of matrix is typically referred to as a phylogenetic distance matrix, and it forms the foundation for many phylogenetic diversity metrics. The phylogenetic distance matrix can be generated using the `cophenetic()` function.

```
> p.dist.mat <- cophenetic(my.phylo)
```

Because the matrix is large (25×25 species) we can look at just the top left corner to get an idea of the output.

```
> p.dist.mat[1:4, 1:4]
```

We see that the resulting matrix has the species names as the row and column names, and the values in the matrix are the sum of the branch lengths separating each pair of species. Thus, if two species are far apart on the phylogeny their distance will be larger than that for two closely related species. The second type of matrix reports the shared branch length between all pairs of species and is often referred to as a phylogenetic variance–covariance (VCV) matrix. This type of matrix is used in some phylogenetic diversity metrics, but it is more commonly used in comparative analyses. A phylogenetic VCV matrix can be computed in R using the `vcv()` function.

```
> vcv(my.phylo)
```

The diagonal values of the matrix are the distances from the root to the tip that contains that species. The off diagonal values indicate the amount of shared branch length between two species. Assuming a Brownian Motion model of trait evolution, the diagonal values are used to estimate the expected variance in a trait, and the off diagonal values are used to estimate the expected covariance in the trait values between two species. I will explain this in more detail in later chapters, but essentially high off diagonal values mean species that are more closely related and are expected to have more similar trait values.

2.5 Simulating Phylogenies in R

Simulation studies are increasingly found in the ecological and evolutionary literature. Some of these studies are strictly experimental in that they experiment with different parameters to observe the likely patterns that can result. Other approaches use simulations to estimate the parameter values that would best explain an observed pattern in an empirical dataset of interest. Both are useful and now widely employed approaches in phylogenetic investigations in ecology and evolution [22–24].

The simulation of phylogenies in R is trivial, but this does not mean such simulations should be used without careful thought. There are a number of ways to simulate phylogenies in R, but we will focus on the two most basic approaches available in the *ape* package. The first approach begins with a single branch that randomly splits into two “daughter” branches. These daughter branches may then branch themselves to produce two daughters and so on until the desired number of terminal nodes is reached. This phylogenetic simulation can be performed in R using the `rtree()` function. Here, we will generate a random phylogeny with random splitting containing 40 species.

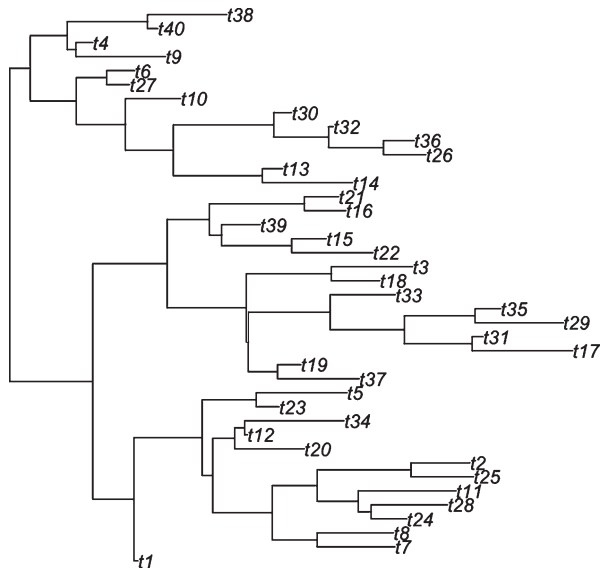
```
> new.tree <- rtree(40, rooted = TRUE, tip.label = NULL)
```

This function can be modified to include a root in the phylogeny or not. The default is to include a root. Now plot your new simulated tree (Fig. 2.10):

```
> plot(new.tree)
```

The first thing you may notice is that the phylogeny is not ultrametric and that it will not look like mine because it was randomly generated. You may also notice that the

Fig. 2.10 A plot of a simulated phylogeny containing 40 species generated by randomly splitting lineages. Note that your phylogeny will be randomly generated and therefore will not look like the phylogeny plotted presently



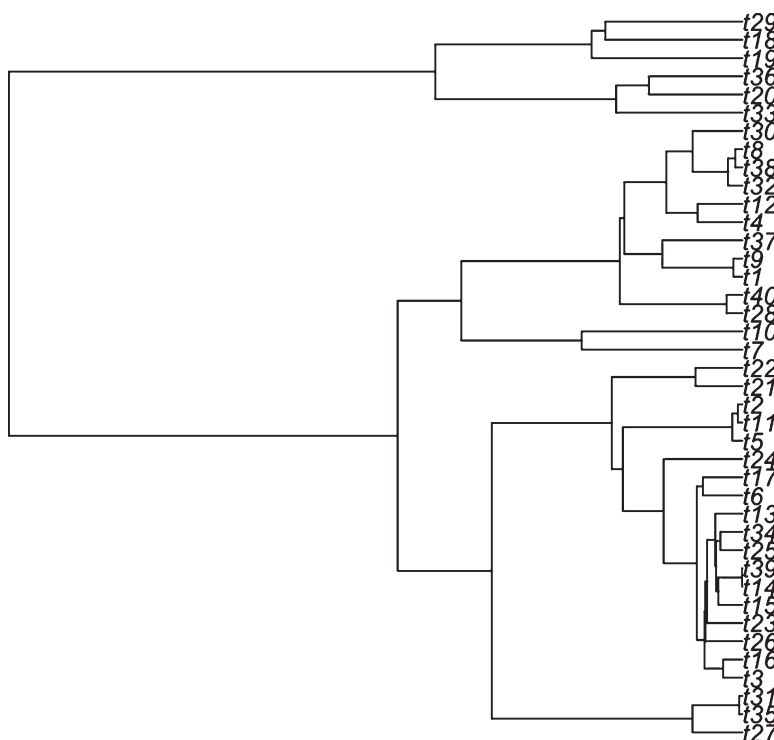


Fig. 2.11 A plot of a simulated phylogeny containing 40 species generated by simulating coalescence. Note that your phylogeny will be randomly generated and therefore will not look like the phylogeny plotted presently

terminal taxa are labeled t_1, t_2, \dots, t_{40} . This nomenclature for terminal taxa will be used in both simulation approaches described here. If you were simultaneously simulating community datasets with 40 taxa with a different naming convention, you could simply replace the names on the phylogeny with your list in the desired order using `new.tree$tip.label <- c(your.vector.of.names)`. If you wished to provide a vector of names prior to the simulation of the phylogeny to simply things, then you could provide that vector after the `tip.label` argument in the function.

The second basic approach for simulating phylogenies in R is to simulate a phylogeny that randomly clusters terminal taxa. Thus, in a sense this approach works backwards towards the tips while the previous method moves in the other direction. This “coalescence” method can be performed using the `rcoal()` function in the *ape* package.

```
> new.ultra.tree <- rcoal(40)
```

This generated a random coalescent tree. Now plot your new simulated coalescent tree (Fig. 2.11):

```
> plot(new.ultra.tree)
```

You will see a rooted and ultrametric phylogeny has been randomly generated. As I mentioned above, these are the two most basic approaches for simulating phylogenies, and if you were interested in performing a simulation study it would be useful to consider the assumptions you would want to make regarding the phylogeny prior to simulating anything and to decide whether these assumptions are met using one of the two approaches above or any other approaches available in R.

Both of the above simulation approaches generate one phylogenetic tree at a time. What if you wanted to generate 100 random phylogenies with 10 species each? You would not want to sit at your computer and enter the line of code 100 times. To perform any function repeatedly in R you can write a loop function called a `for()` loop. Such loops are generally fine for computationally easily problems like simulating a phylogeny with ten species, but for larger problems the `replicate()` function may be quicker. Though for simplicity we will just use a `for()` loop here and only produce nine random splitting phylogenies with ten terminal taxa in each and write each phylogeny to your working directory with a unique file number with a “.txt” file ending. If you are new to R you will see lines of code starting with a “#” symbol. This means that the text in that line following the symbol is a comment which R will disregard. Commenting your code is essential particularly when writing loops or functions. This helps you understand what you are doing and your goals and it also greatly helps anyone that may use your code in the future.

```
> for(i in 1:9){
  ## Make a random tree with 10 terminal taxa
  temp.random.tree <- rtree(10)

  ## Write the random tree to your working
  ## directory with each new file having a new
  ## number
  write.tree(temp.random.tree, paste(i, ".txt",
    sep=""))

  ##close the loop
}
```

If you now navigate to your working directory on your hard drive you will see nine new files named “1.txt,” “2.txt,” ..., “9.txt.” Open one of the files in a text editor and you will see a Newick version of your simulated random splitting phylogenetic tree for that iteration of the `for()` loop. Thus, it can be fairly easy and fast to simulate a phylogeny in R and again the central issue will be whether the simulation you are using is appropriate for your particular research goals. It is not possible for me to recommend one simulation approach over another for every problem because the goals of simulation approaches vary from study to study. Thus, my goal here is to simply introduce you to two basic simulation approaches to get you started and to demonstrate that simulating phylogenetic data is quite simple in R.

2.6 Conclusions

We have now covered the basics of phylogenetic information in R particularly focusing on the “phylo” class. There is a much more extensive text in the UseR! series on phylogenetic information in R that I recommend for more detail [15], but for the analyses in the present book this chapter serves as a sufficient primer. The most important aspects from the present chapter that you should keep at the forefront of your mind as we proceed into the next chapter on phylogenetic diversity have to do with how branch length information is stored and manipulated in the “phylo” class.

2.7 Exercises

1. Make a Newick file for five species (speciesA, speciesB, speciesC, speciesD, and speciesE) where speciesA and speciesE are most closely related to one another, speciesB and speciesD are most closely related to one another, and speciesC is most closely related to speciesA and speciesE. The file should have no branch lengths. Read this file into R and plot it with blue branches.
2. Take the Newick file you just made and put branch lengths in the file. Set all branch lengths to be 3.00 units in length and calculate the total branch length of the phylogeny (i.e., the tree length).
3. Calculate all subtrees for your phylogeny. Next choose one subtree and calculate the length of all the branches of that subtree and divide this number by the sum of the branch lengths in the whole phylogeny.
4. Use the `sample()` function to randomize the tip labels on a phylogeny where no name is lost and no name is used twice in the new phylogeny (i.e., sample without replacement).
5. Take the Newick file you made in Exercise 2 above and print a Nexus version of the file to your R console.
6. Repeat 1–5 above, except first generate a six species phylogeny speciesA–speciesF. speciesA and speciesB should be most closely related to one another, speciesC and speciesD should be most closely related to one another, and speciesE and speciesF should be most closely related to one another. Finally, speciesE and speciesF should be more closely related to speciesA and speciesB and more distantly related to speciesC and speciesD.
7. Simulate 45 coalescent trees (10 species in each) and write them to your working directory.
8. Put the 45 random coalescent tree files into a new directory, change your R working directory to that new directory and read each phylogeny into R, and calculate the total tree length for each automatically (i.e., do *not* do this one tree at a time—automate it). You will need to use the `list.files()` command and make a `for()` loop.

<http://www.springer.com/978-1-4614-9541-3>

Functional and Phylogenetic Ecology in R

Swenson, N.

2014, XII, 212 p. 36 illus., 2 illus. in color., Softcover

ISBN: 978-1-4614-9541-3