Evolutionary Trees

Fredrik Ronquist

August 29, 2005

1 Evolutionary Trees

Tree is an important concept in Graph Theory, Computer Science, Evolutionary Biology, and many other areas. In evolutionary biology, a tree is a branching diagram used to illustrate the divergence over time of evolutionary lineages (species) or of genetically encoded traits (such as DNA or protein sequences) of those lineages. The former trees are sometimes called *species trees* and the latter *gene trees*, with the word *gene* loosely referring to any genetic component or genetically determined trait of an organism. The word *phylogeny* is usually used as a synonym of species tree.

In organisms that reproduce asexually, gene trees and species trees are essentially the same thing but organisms that reproduce sexually, including most multicellular organisms, mix genes with different ancestry every time they reproduce. Therefore, sexually reproducing organisms diverge more slowly than their genes, and this introduces several potential sources of mismatch between gene and species trees, as we will see later during the course. Generally speaking, however, gene trees and species trees tend to be largely congruent.

We will, during the course, cover both Population Genetics and Phylogenetics, in particular the ways in which they use evolutionary trees. *Population Genetics* focuses on individual or closely related evolutionary lineages and the genetic structure within them. This information, contained in gene trees, is used to describe the present, infer the recent past, or predict the immediate future of evolutionary lineages (populations). *Phylogenetics* is the study of species trees and their implications concerning the evolution over longer time scales. There is an interesting divide between Population Genetics and Phylogenetics, among other things evident in the fact that evolutionary lineages are referred to as populations in the former field and species in the latter. The critical difference appears to be in the life span of the lineages: the term *species* implies a longer life span than the term *population*. A lineage that lasted or is predicted to last some critical amount of time (say 10,000 to 100,000 years or more) is typically referred to as a species, whereas a lineage with shorter life span is called a population and not a species. There is also some anthropocentrism involved, so we tend to call lineages closely related to us separate species (eg. Neanderthals) whereas similar amounts of divergence would probably be understood as population-level differentiation in invertebrates.

An evolutionary tree is typically drawn with the root at the bottom (Fig. 1) unlike other types of trees, such as those used in Computer Science, which by convention are drawn with the root at the top. The evolutionary tree (or any other tree for that matter) consists of two components: the divergence points and the lines connecting them. Biologists typically refer to the divergence points as *nodes* and the lines connecting them as *branches* or sometimes *internodes* (Fig. 1).

Figure 1: An evolutionary tree

The nodes in an evolutionary tree fall into three types: terminal nodes, interior nodes, and the root node (Fig. 1). The *terminal nodes* either represent current lineages or lineages that have gone extinct (evolutionary dead ends). The *interior nodes* represent slightly different things depending on whether the tree is a species tree or a gene tree. If it is a species tree, then an interior node symbolizes a *speciation* event whereby one *ancestral (mother)* lineage diverges into two *descendant (daughter)* lineages. The two daughter lineages are also referred to as *sister* lineages. Note that biologists tend to use feminine rather than masculine (father) or neutral (children, siblings) words to describe these tree components. If the tree is a gene tree in a sexually reproducing organism, then the interior nodes are genetic divergence events that ultimately go back to a DNA or RNA replication event. Some of those events may correspond to but typically precede speciation events, as will be explained later during the course.

The *root node* indicates where the tree is rooted, that is, the position of the most basal divergence event in the tree. If the tree is a species tree, then the root is the place where it connects to the rest of the *Tree of Life*, the species tree for all life on earth. The root of a species tree is also referred to as the *most recent common ancestor* of the species in the tree. This term implies that the node represents the last snapshot of the ancestral lineage just before the divergence event. We can similarly think of an interior node in a species tree as symbolizing either a snapshot of the ancestral linage just before the speciation (an *ancestor*) or the speciation event itself. The nodes in a gene tree are typically not described as ancestors but sometimes as ancestral sequences (or traits).

The branches in an evolutionary tree have lengths measured in time units. This means that (in a tree with the root at the bottom) we can place a time scale along the tree and the vertical position of the divergence events will then indicate when they took place (Fig. 1). Similarly, the vertical range of a branch in such a tree corresponds to the life span of the corresponding evolutionary lineage. In a tree with slanted branches (Fig. 1), the actual length of a branch also depends on its angle, which is typically chosen to give a graphically pleasing presentation, and therefore has no direct interpretation. To avoid this problem, evolutionary trees are often drawn with rectangular branches (Fig. 2). In such a tree, each branch is represented by two lines; the length of one reflects the time duration of the corresponding lineage and the length of the other is chosen so that the branch can be connected with its ancestral lineage.

It is important to understand that evolutionary trees are only models, even if they are usually very good models (hence their use in virtually all disciplines in the life sciences). There are several processes that violate the basic assumptions of the tree model, including recombination (common in sexually reproducing organisms) and horizontal gene transfer (rare but occurring in most organisms). For now, we will ignore these processes but we will return to them later during the course. An important fact to remember is that most evolutionary lineages are short-lived and go extinct before they become significant enough to be considered populations or species. There is usually an implicit understanding that an evolutionary tree is what remains after a large number, perhaps

Figure 2: An evolutionary tree with rectangular branches.

millions, of such insignificant side branches have been pruned away.

2 More About Trees

An evolutionary tree is a special case of the general concept of trees, and it is characterized by several special properties. First, it is *rooted*, that is, it has a root. Another way of saying the same thing is that the tree is *directed*, there is an explicit time direction in it (otherwise we would not be able to identify ancestors and descendants). Second, the tree is a *clock tree*, meaning that branch lengths are measured in time units. A clock tree is sometimes also called a *linearized tree*, an *ultrametric tree*, or a *(labeled) history*.

In practice, it is seldom possible to directly infer an evolutionary tree of the type described above, and we will often use the term evolutionary tree to refer to simpler models. In particular, it is difficult to estimate the divergence times because of variation in evolutionary rates. For this reason, we often have to be satisfied with trees where branch lengths are measured in terms of the amount of evolutionary change instead of in terms of time. Such a tree is referred to as a *non-clock tree* or an *additive tree*. If we consider a species tree for a set of extant (now living) species, where all terminals are of the same age, then a clock tree has all terminals at the same level (Fig. 3a) whereas a non-clock tree (potentially) has the terminals at different levels (Fig. 3b). In the non-clock tree, the terminals on short branches are more similar to the most recent common ancestor of the group than the terminals that sit on long branches. Biologists sometimes refer to additive trees as *phylograms*.

Figure 3: Comparison between a non-clock tree (a) and a clock tree (b).

As we will discover soon, many methods of inferring evolutionary trees cannot estimate divergence times nor can they place the root in the tree. Such a method will result in an *unrooted tree*, also called an *undirected tree* because of its lack of a time direction (Fig. 4a). An unrooted tree is always non-clock. If you are interested in the position of the root, which is almost always the case, you have to use a separate rooting method to find it. A common method is to include one or more reference lineages, also called *outgroups*, in your analysis. After the analysis, the root is placed between the outgroup and the lineage being studied (the *ingroup*) (Fig. 4b).

Biologists are sometimes just interested in the branching structure or branching order of an evolu-

Figure 4: An unrooted tree (a) can be rooted between an outgroup and the study group (the ingroup) (b) to give a rooted tree (c).

tionary tree, and not in the branch lengths. The branching structure is referred to as the *topology* of a tree and is represented in a *cladogram*. The same cladogram can be drawn in many different ways (Fig. 5).

Evolutionary biologists typically restrict their attention to trees that are binary or dichotomous, meaning that there are only two descendants of each interior node. Sometimes, trees with three or more descendants from a single node, called *polytomous trees*, are also considered. In many situations, polytomous trees are adequately approximated by sets of binary trees with short interior branches (Fig. 6).

Regardless of its type (clock, rooted non-clock or unrooted) it is always possible to draw the same tree in different ways in a two-dimensional figure because of the rotational freedom around each node. For instance, in a rooted tree we can shift the left and right descendants of each node without changing the meaning of the tree (Fig. 7).

In Graph Theory and Computer Science, there is a different set of terms used for trees, and you

Figure 5: The same tree topology (cladogram) can be drawn in many different ways because, among other things, the branch lengths are arbitrary.

often see these used in computational evolutionary biology as well. They are *vertex* (for node), *edge* (for branch), *leaf* (for terminal node), and *root* (for the root node). In this terminology, the *degree* of a vertex is the number of branches connected to it. Thus, an interior node of a binary tree has degree three, a leaf has degree one and the root has degree two. An *edge weight* is the same as the length of a branch.

It is often practical to refer to subtrees within a larger tree. Biologists refer to them as *clades.* A clade can defined as a node i and all of its descendants (or, alternatively, as the subtree rooted at node i). Clades can also be called *taxa* (singular taxon) or *operational taxonomix units (OTUs).* Taxa are used for named groups in a biological classification whereas OTUs are groups that lack formal names. It is not uncommon to see evolutionary trees in which the terminals are taxa above the species level (also called higher taxa). For instance, a cladogram may give the relationships among a set of genera or families, each containing many species. Since individual evolutionary lineages can also be called taxa or OTUs, it is generally true that the terminals in a species tree are taxa or OTUs.

Figure 6: A tree with a polytomy (a) can be thought of as being approximated by a set of of binary trees with very short interior branches (b to d).

Closely related to the concept of a subtree is the concept of a *split* or taxon *bipartition*. Each branch in a tree divides the terminal nodes (taxa) into two disjoint sets, one on each side of the branch. We will be using splits when we describe methods for assessing the robustness in the inference of evolutionary trees.

3 Computer Representation of Trees

Reference: Felsenstein Chapter 35, pp. 585-590.

Much of this course will be devoted to the description of algorithms operating on trees. For this purpose, we will need a data model for trees. A simple model for rooted binary trees, found in many computer science texts, is given below (Fig. 8; see also Felsenstein, Fig.35.1). We will be referring to this model as the *Binary Tree Data Model* (BTM). The idea is to use a simple data structure for each node, containing pointers to the two descendant nodes and the ancestral node.

Figure 7: The same tree can be drawn in many different ways because of the rotational freedom around each node.

The terminal nodes have their descendant node pointers set to NULL and the root node has its ancestral node pointer set to NULL. Strictly speaking, the pointer to the ancestral node is not needed but it is convenient to have it in many cases. Felsenstein also includes a boolean variable indicating whether a node is terminal (a tip). Again, this can be figured out from the pointers, so this variable is only there for convenience. If we need to store branch lengths, we can simply store them in the node structure of the node above the branch. The branch length variable needs to be added to the node structure in Felsenstein's figure. In an object oriented computer language, we would typically store the nodes inside a tree class, which would contain a pointer to the root of the tree.

A complication arises when we want to represent unrooted trees using this structure, since they do not have a root node nor a natural ancestor - descendant direction to them. One way of solving the problem is to treat the tree as if it were rooted on an internal node, typically the interior node adjacent to the outgroup terminal (Fig. 9). We can easily represent all of the nodes that are descendants of the interior 'root' node using the BTM. The interior root node is then connected to

Figure 8: The Binary Tree Data Model (BTM).

two of its subtrees using its left and right pointers, and the third subtree using its ancestor pointer. All the branch lengths around the interior root node are now stored in other nodes, so the interior root node does not have a branch length. By using this type of structure, we have forced a direction onto the unrooted tree.

Clearly, truly polytomous trees cannot be represented using the BTM. A data structure similar to the BTM that accommodates polytomous trees uses pointers from a node to its ancestor, leftmost descendant, and to one of its siblings (Fig. 10). This structure can easily accommodate unrooted trees by arbitrarily selecting one of the interior nodes as the 'root' of the tree. We will refer to this data structure as the *Polytomous Tree Data Model* (PTM).

A somewhat different structure has been pioneered by Felsenstein (Fig. 11; see also Felsenstein: Fig. 35.2) and will be referred to as the *Felsenstein Tree Data Model* (FTM). It uses rings of nodelets to represent nodes. Each nodelet has a 'next' pointer to the next node in the ring and an 'out' pointer to the nodelet on the other end of the branch to which it is connected. This structure can accommodate both binary and polytomous trees as well as rooted and unrooted trees. An

Figure 9: A modification of BTM to accommodate unrooted trees.

advantage of this data structure is that it can be rerooted more easily than the previous two types. Branch lengths can either be stored in separate data structures pointed to by both of the adjacent nodelets, or they can be contained as member variables of both of the adjacent nodelets, and set functions can then be used to make sure that the two member variables representing the same branch are always in sync. It is convenient to have the nodelet structure or class contain a boolean member indicating whether the nodelet points down in the tree, so that we know where to start and stop when cycling through the nodelets representing a node of the tree.

4 Text and XML Representation of Trees

Reference: Felsenstein Chapter 35: pp. 590-591.

There is often a need to describe a tree in text format, for instance for storing trees resulting from an analysis. This is typically done using the Newick format described in Felsenstein (Chapter 35, pp. 590). A shortcoming of the Newick format is that it can only specify rooted trees. An unrooted

Figure 10: The polytomous tree data model (PTM).

tree must be converted to one of its rooted representations before it can be expressed in Newick format. If we arbitrarily root an unrooted tree on one of its branches, however, there is one branch length that will have to be arbitrarily divided into two components, one on each side of the root (Fig. 9a). A better method is to root the tree on an interior node and write it as if it had a basal trichotomy (since the interior 'root' node will have three 'descendants' unlike all other nodes, which have only two). If there is a single outgroup terminal, the tree is typically rooted on the interior node adjacent to that terminal. Assume we have an unrooted binary tree with terminals labeled A to D and the branch lengths labeled v_1 to v_5 and that we root it on the interior node adjacent to the tip D (alternatively, we can describe this as rooting the tree on D) (Fig. 9b). The Newick-format description would now be ((A:v_1,B:v_2):v_3,C:v_4,D:v_5), as if the tree had a basal trichotomy involving C, D, and the ancestor of A and B. Note how this solution gives us a natural place to put all branch lengths. The basal trichotomy solution only works if we restrict our attention to binary trees, since there is no way we can distinguish a truly rooted tree with a basal trichotomy from an unrooted tree written as if it were a rooted tree with a basal trichotomy.

A Newick-format tree is often contained in a tree block of a NEXUS file. The general structure of

Figure 11: The Felsenstein tree data model (FTM).

a NEXUS file containing a tree block is:

```
#NEXUS
begin trees;
    translate
        1 Man,
        2 Chimp,
        3 Gorilla,
        4 Orangutan,
        5 Gibbon;
    tree example = (((1,2),3),4,5);
end;
```

Note the translate command, which allows you to map some tokens in a Newick-format tree description (typically integers) to real taxon (terminal node) names. Note also that the tree is given without branch lengths; this is permitted by the Newick format. In a commonly used extension of the Nexus tree format, we can also specify whether the tree is rooted or unrooted using the tags [& R] (for rooted) and [& U] (for unrooted). The tree above would then be given as

```
tree example = [&U] (((1,2),3),4,5);
```

to indicate that it should be understood as an unrooted tree (instead of a rooted tree with a true basal trichotomy).

As Felsenstein mentions, it is likely that the Newick format will be replaced, or at least complemented, by an XML standard for describing trees. The XML code suggested by Felsenstein seems reasonable to us but since there are a couple of errors in his example, we reiterate it here for clarity (hopefully without errors):

```
<phylogeny>
    <clade>
        <clade length=0.06>
            <clade length=0.102><name>A</name></clade>
            <clade length=0.23><name>B</name></clade>
        </clade>
        <clade length=0.4><name>C</name></clade>
    </clade>
</phylogeny>
```

Note the use of the tag 'clade' for subtrees and 'phylogeny' for the entire tree. To preserve more of the NEXUS format, one could perhaps have used the term 'tree' instead of 'phylogeny'. The phylogeny (or tree) tag would presumably be able to hold information on whether the tree is rooted or unrooted, and the phylogeny could also contain a name tag if the tree is named.

5 Some Simple Tree Algorithms

Reference: Felsenstein Chapter 35: pp. 587-589.

Most algorithms on trees involve repeating the same operation on each node (or branch) of the tree in turn, either passing from the tips of the tree down to the root or from the root up to the tips. Biologists often refer to these traversals as the *downpass* and the *uppass* sequence, respectively. In computer science they are known as the *postorder* (downpass) and *preorder* (uppass) traversals, respectively. There is also an inorder traversal but it is rarely used for evolutionary trees.

Algorithm 1 Postorder traversal algorithm

```
Traverse left descendant of $p$
Traverse right descendant of $p$
Carry out $f(p)$ on $p$
```

Assume that, for each node p, we want to implement a function $f(p)$. Then the postorder traversal is easily implemented as a recursive function (Algorithm 1). It is called postorder traversal because the function is carried out *after* passing through the subtree rooted at p. The preorder traversal algorithm is similar (Algorithm 2) but the function is now carried out *before* passing through the subtree rooted at p.

Algorithm 2 Preorder traversal algorithm

```
Carry out $f(p)$ on $p$
Traverse left descendant of $p$
Traverse right descendant of $p$
```

If you want to minimize the use of recursive algorithms, you can generate an array of node (or branch) pointers in postorder traversal order using a recursive function once. Anytime you need a postorder traversal of the tree after this initial call, you can then simply loop over the elements of the downpass node array in order. Similarly, a preorder traversal is implemented by looping over the elements in the downpass array in reverse order.

The following code illustrates how you can create a postorder traversal array using Java (assuming rooted trees represented using BTM):

```
public class Tree {
    public Node [] downPass;    // use as pointers
    public Node theRoot;        // use as pointer

    private int index;
    private Node [] theNodes;   // this is where we actually keep the nodes

    // code to construct the tree and set theRoot appropriately

    public void GetDownPass (void) {
        index = 0;
        DownPass (theRoot);
    }

    private void DownPass (Node p) {
        if (!p.tip)
        {
            DownPass (Node p.leftdesc);
            DownPass (Node p.rightdesc);
        }
        downPass[index++] = p;
    }
}
```

The downPass array can then be accessed by using code such as

```
for (int i=0; i<theTree.GetNumNodes(); i++) {
    Node p = theTree.allDownPass[i];
    // Do something with the node p here
}
```

Note that the `downPass[]` array is made public here, which is convenient but exposes the array to users of the tree class. Better encapsulation can be provided, for instance, by using get functions to access the elements of the `downPass[]` array. In many algorithms we need to handle only internal nodes, so it makes good sense to keep postorder arrays of both all the nodes and only the interior nodes in the tree (including the root).

To print a tree in Newick format, it is convenient to use a recursive algorithm for printing a subtree rooted at p and call it initially with the root of the tree. If we use the BTM and the convention described above for storing ordered and unordered binary trees, the algorithm needs to include an extra branch for the 'root' of the tree (Algorithm 3). Note that the algorithm will not print the semicolon required after the parenthetical description of the tree, so this element will have to be added.

Almost anything you can do in a recursive algorithm can also be done using an ordinary algorithm. For instance, you can print a BTM unrooted or rooted tree using a non-recursive algorithm if you add an integer member (let's call it x_p) to each node p and use that to indicate whether the node is

Algorithm 3 Recursive algorithm for printing a tree (BTM unordered or ordered)

```
if $p$ is not tip then
  Print ‘(’ and left subtree of $p$
  Print ‘,’ and right subtree of $p$
  if $p$ is ‘root’ of unrooted tree then
    Print ‘,’ and ancestor subtree of $p$
  end if
  Print ‘)’
end if
Print name of $p$ (if any)
Print ‘:’ and branch length of $p$ (if any)
```

being visited the first, second, or third time; the ‘root’ of an unrooted tree will actually be visited also a fourth time (Algorithm 4). The algorithm starts with p being the root node (ρ) of the tree.

To construct a binary rooted or unrooted BTM tree from a Newick-format string, you can use a recursive algorithm for reading a subtree rooted at node p (Algorithm 5). It uses a global index variable (i), which indicates the position from which we are currently reading the string or array s containing the Newick-format tree. You start the algorithm by setting $i = 0$ and calling it with the root of the tree.

Again, the same task can be accomplished using a non-recursive algorithm (Algorithm 6). It uses a semicolon to mark the end of the string or array containing the tree description; any other way of determining the end of the tree description would work equally well.

If you are interested in exploring the PTM, here is a snippet of Java code indicating the basic recursive algorithm for a postorder traversal:

```
public void Traverse (Node p) {
    Node q;
    for (q = p.left; q != null; q=q->left) {
        Traverse (q);
    }
    F(p);
}
```

The same code could be written like this using the FTM (assuming that a tip node has an out

Algorithm 4 Nonrecursive algorithm for printing a tree (BTM unordered or ordered)

```
for all $p$ do
  $x_p \leftarrow 0$
end for
$p \leftarrow \rho$
while $p \neq \emptyset$ do
  $x_p \leftarrow x_p + 1$
  if $p$ is not tip then
    if $x_p = 1$ then
      Print '('
      $p \leftarrow$ left descendant of $p$
    else if $x_p = 2$ then
      Print ','
      $p \leftarrow$ right descendant of $p$
    else if $x_p = 3$ and $p$ is 'root' of unrooted tree then
      Print ','
      $p \leftarrow$ ancestor of $p$
    else if $x_p \geq 3$ then
      Print ')'
    end if
  end if
  if $x_p \geq 3$ or $p$ is tip then
    Print name of $p$ (if any)
    Print ':' and branch length of $p$ (if any)
    if $x_p > 3$ ('root' of unrooted tree) then
      $p \leftarrow \emptyset$
    else
      $p \leftarrow$ ancestor of $p$
    end if
  end if
end while
```

pointer pointing to itself):

```
public void Traverse (Node p) {
    Node q;
    for (q = p.next; q != p; q=q->next) {
```

Algorithm 5 Recursive algorithm for reading a tree (BTM unrooted or rooted)

```
if $s[i]$ is ‘(’ then
    $i \leftarrow i + 1$
    Create a new node $q$
    Set left descendant of $p$ to $q$
    Set ancestor of $q$ to $p$
    Read subtree rooted at $q$
    Check that $s[i]$ is ‘,’
    $i \leftarrow i + 1$
    Create a new node $q$
    Set right descendant of $p$ to $q$
    Set ancestor of $q$ to $p$
    Read subtree rooted at $q$
    if $p$ is ’root’ of unrooted tree then
        Check that $s[i]$ is ‘,’
        $i \leftarrow i + 1$
        Create a new node $q$
        Set ancestor of $p$ to $q$
        Set ancestor of $q$ to $p$
        Read subtree rooted at $q$
    end if
    Check that $s[i]$ is ‘)’
    $i \leftarrow i + 1$
end if
Read name of $p$ (if any)
Read ‘:’ and branch length of $p$ (if any)
Set $i$ to index of next character to read
```

```
            Traverse (q.out);
        }
        F(p);
}
```

Both models give quite compact code for reading a tree since we do not have to deal with any special cases for unrooted trees. The recursive tree writing algorithm could be written like this for PTM:

Algorithm 6 Nonrecursive algorithm for reading a tree (BTM unrooted or rooted)

```
$i \leftarrow 0$
while $s[i]$ is not ';' do
  if $s[i]$ is '(' then
    $i \leftarrow i + 1$
    Create a new node $q$
    Set left descendant of $p$ to $q$
    Set ancestor of $q$ to $p$
    $p \leftarrow q$
  else if $s[i]$ is ',' then
    Create a new node $q$
    if $p$ is left descendant of its ancestor then
      Set right descendant of ancestor of $p$ to $q$
    else
      Set ancestor of ancestor of $p$ to $q$
    end if
    Set ancestor of $q$ to $p$
    $p \leftarrow q$
  else if $s[i]$ is ')' then
    Set $p$ to ancestor of $p$
  else
    Read label of $p$ (if any)
    Read ':' and branch length of $p$ (if any)
    Set $i$ to index of next character to read
  end if
end while
```

```
String treeDescription = "";

public void PrintSubtree (Node p) {
    if (!p.tip) {
        treeDescription += "(";
        for (Node q = p.left; q != null; q = q.sib) {
            PrintSubtree (q);
            if (q.sib != null)
                treeDescription += ",";
        }
        treeDescription += ")";
    }

    if (p.tip)
        treeDescription += p.GetLabel();

    if (hasBrlens && p != theRoot) {
        treeDescription += ":";
        treeDescription += p.GetLength();
    }
}
```

Note the clean code due to the lack of special cases. The same algorithm might look almost identical for FTM:

```
String treeDescription = "";

public void PrintSubtree (Node p) {
    if (!p.tip) {
        treeDescription += "(";
        for (Node q = p.next; q != p; q = q.next) {
            PrintSubtree (q.out);
            if (q.next != p)
                treeDescription += ",";
        }
        treeDescription += ")";
    }

    if (p.tip)
        treeDescription += p.GetLabel();

    if (hasBrlens && p != theRoot) {
        treeDescription += ":";
        treeDescription += p.GetLength();
    }
}
```

As stated previously, the FTM representation has the advantage that it can be rerooted more easily than BTM and PTM. However, it is also fairly common that we want to pass from a node *p* down to the root. This is easily done with BTM or PTM using code such as:

```
while (p != theRoot) {
    // do something with p
    p = p.anc;
}
```

The FTM representation, on the other hand, requires slightly more complex code:

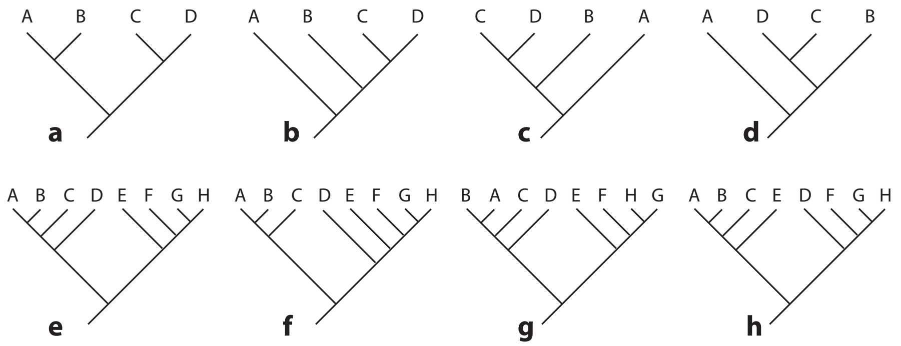
```
while (p != theRoot) {
    // do something with p

    p = p.out;

   // assume we have a boolean called top marking whether the
   // node is on the top of a branch
   while (!p.top)
        p = p.next;
}
```

6 Study Questions

1. Define a clock tree, a non-clock tree, a rooted tree, and an unrooted tree.
2. Define terminal node, interior node, root, and branch.
3. Can a clock tree be unrooted? Can a non-clock tree be rooted?
4. How many branches are there in a rooted binary tree with n terminals? How many interior nodes are there (count the root as an interior node)?
5. How many branches are there in an unrooted binary tree with n terminals? How many interior nodes are there?
6. What is the difference between a cladogram and a phylogram?

7. Of the rooted trees given below (Fig. 12), which have identical topology? Which trees have identical topology after the root is removed?

Figure 12: Some rooted trees. Although all of them look different, some of them specify the same topology



8. Give a recursive algorithm performing the function $f(p)$ in postorder sequence on the nodes p of a tree represented by the PTM.

9. Describe a complete data structure for the FTM using Java code.

10. Show how a postorder array of nodes for the BTM can be used to perform a function $f(p)$ on each node of a tree in a preorder traversal.