

11

Trees

Introduction

This chapter is devoted to exploring the computational aspects of the study of trees. Recall from the textbook that a tree is a connected simple graph with no simple circuits.

First, we will discuss how to represent, display, and work with trees using *Mathematica*. Specifically, we will see how to represent rooted trees and ordered rooted trees in addition to simple trees. We then use these representations to explore many of the topics discussed in the textbook. In particular, we will see how to use binary trees to store data in such a way as to make searching more efficient and we will see an implementation of Huffman codes. We will use *Mathematica* to carry out the different tree traversal methods described in the text. We will see how to construct spanning trees using both depth-first and breadth-first search and how to use backtracking to solve a variety of interesting problems. Finally, we will implement Prim's algorithm and Kruskal's algorithm for finding spanning trees of minimum weight for a weighted graph.

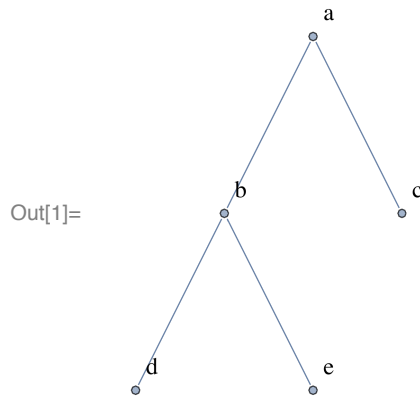
11.1 Introduction to Trees

In this section we will focus on how to construct trees in *Mathematica* and how to check basic properties, such as determining if a tree is balanced. To begin, we will consider the simplest case, unrooted trees, before moving on to rooted and ordered trees.

Unrooted Trees

Recall that a tree is defined to be a graph that is undirected, connected, and has no simple circuits (or cycles). To create a tree, we can just create a Graph as we did in the previous chapter.

```
In[1]:= firstTree = Graph[{ "a" → "b", "a" → "c", "b" → "d", "b" → "e"},
    DirectedEdges → False,
    VertexLabels → "Name", ImagePadding → 10]
```



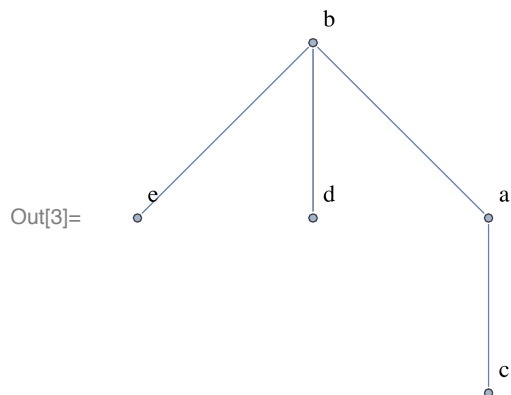
The first thing you may notice is that *Mathematica* has automatically drawn this in the traditional way, which suggests that *Mathematica* recognized the graph as a tree. The [TreeGraphQ](#) function can be used to check if a graph is a tree.

```
In[2]:= TreeGraphQ[firstTree]
```

Out[2]= True

Note that the order of the vertices can affect which vertex *Mathematica* draws at the top of the tree. For example, in the tree above, we can encourage *Mathematica* to draw the vertex *b* at the top of the tree by giving a list of the vertices as the first argument. We list the vertices in layered order, that is, the top vertex will be first, followed by the vertices we wish to see in the second layer, then the vertices in the third layer and so forth. Note that changing the order of vertices within a layer can affect their horizontal position.

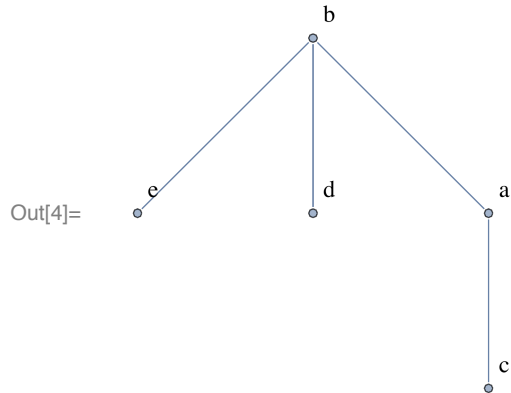
```
In[3]:= Graph[{ "b", "e", "d", "a", "c"},
    {"a" → "b", "a" → "c", "b" → "d", "b" → "e"},
    DirectedEdges → False, VertexLabels → "Name", ImagePadding → 10]
```



There is a limit to the amount of influence this provides, however. For example, *Mathematica* has a strong preference for shorter, more balanced, trees. Attempting to use the vertex order to create a

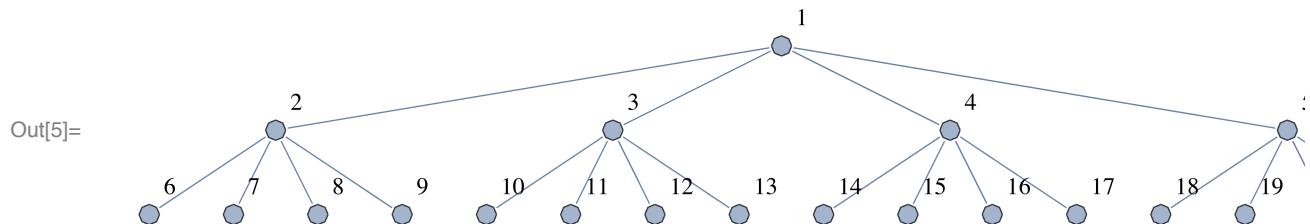
deeper tree will often be ignored. For example, below we attempt, and fail, to get e at the top of the tree.

```
In[4]:= Graph[{"e", "b", "d", "a", "c"},
  {"a" → "b", "a" → "c", "b" → "d", "b" → "e"},
  DirectedEdges → False, VertexLabels → "Name", ImagePadding → 10]
```



Mathematica provides functions for creating certain kinds of trees. Recall that a rooted tree is called k -ary if every vertex has no more than k children. When $k = 2$, we say that it is a binary tree. Also recall that the height of a rooted tree is the maximum number of levels in the tree, i.e., the height is the length of the largest path from the root to any other vertex. (*Mathematica* says that the tree has h levels, which is synonymous with height h .) Given the height as the first argument and k as the second, the function CompleteKaryTree produces the complete k -ary tree of height h . If the second argument is omitted, it defaults as 2, producing a complete binary tree of the specified height.

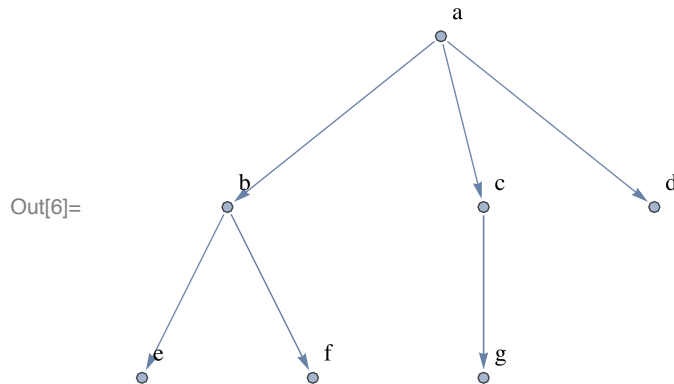
```
In[5]:= CompleteKaryTree[3, 4, VertexLabels → "Name", ImagePadding → 5]
```



Rooted Trees

Next we consider rooted trees. Recall that a rooted tree is a directed graph whose underlying undirected graph is a tree and in which one vertex is designated as the root and all edges are directed away from the root. For example, the following graph is a rooted tree.

```
In[6]:= firstRooted =
  Graph[{"a" → "b", "a" → "c", "a" → "d", "b" → "e", "b" → "f",
    "c" → "g"}, VertexLabels → "Name", ImagePadding → 10]
```



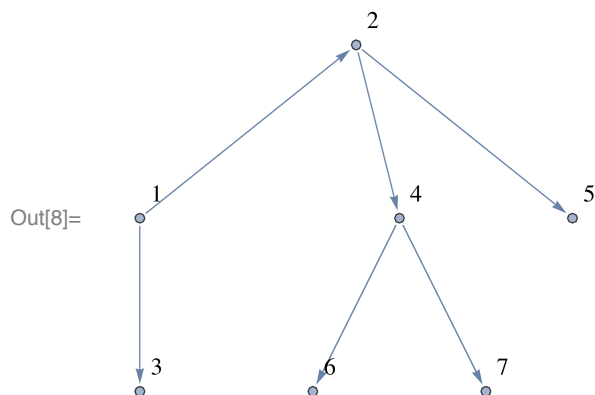
Note that *Mathematica* drew this graph as a tree, just as you would hope. In fact, *Mathematica* does not make a distinction between directed edges and undirected edges for trees, and TreeGraphQ will identify this as a tree.

```
In[7]:= TreeGraphQ[firstRooted]
```

Out[7]= True

While *Mathematica* does recognize trees with directed edges, it does not identify a root. When drawing the tree, *Mathematica*'s emphasis is on minimizing the height, as the following example illustrates.

```
In[8]:= secondRooted = Graph[{1 → 2, 1 → 3, 2 → 4, 2 → 5, 4 → 6, 4 → 7},
  VertexLabels → "Name", ImagePadding → 10]
```

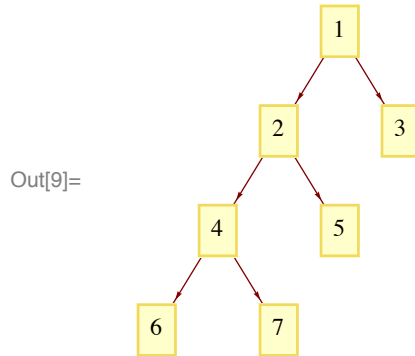


Mathematica has drawn this tree with 2 at the top of the image, despite the fact that it does not satisfy the definition of a root, as it has an edge directed towards it.

To draw the tree as intended, we can apply TreePlot. This is related to the GraphPlot function discussed in the previous chapter, used for multigraphs. TreePlot accepts up to three arguments. The first argument is a list of edges that form the tree. The first argument can also be a Graph object. The second argument, which is optional, specifies where the root of the tree should be drawn. Possible positions are: Top, Bottom, Left, Right, Center. The third argument, which is also optional but

can only be used when the second is present, specifies one of the vertices in the tree to be the root. After all arguments are given, you may use the same options as described for `GraphPlot` in Chapter 10. The following illustrates how to draw **secondRooted** with the root 1 in its proper position.

```
In[9]:= TreePlot[secondRooted, Top, 1, VertexLabeling -> True]
```

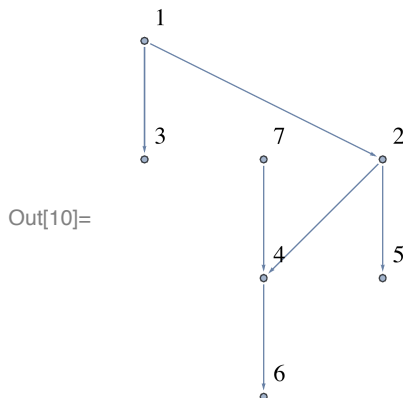


Identifying Roots and Rooted Trees

As mentioned above, *Mathematica*'s `TreeGraphQ` will tell you if a graph is a tree. For directed graphs, it will return false if the graph is not a rooted tree. However, *Mathematica* does not include a function for finding the root of a rooted tree.

We begin by creating an example of a directed graph which is not a rooted tree, but whose underlying undirected graph is a tree.

```
In[10]:= notRooted = Graph[{1 -> 2, 1 -> 3, 2 -> 4, 2 -> 5, 4 -> 6, 7 -> 4},
  VertexLabels -> "Name", ImagePadding -> 10]
```



This is identical to **secondRooted**, except for the direction of the edge between vertex 4 and vertex 7. Note that `TreeGraphQ` identifies it correctly.

```
In[11]:= TreeGraphQ[notRooted]
```

Out[11]= False

However, the underlying undirected graph is a tree.

```
In[12]:= TreeGraphQ[UndirectedGraph[notRooted]]
```

```
Out[12]= True
```

It will be useful to be able to distinguish, however, between a rooted tree and a non-rooted tree. That is, we will need a function that returns True for **firstRooted**, but False on its undirected underlying graph.

```
In[13]:= rootedTreeQ[T_Graph] := TreeGraphQ[T] &&
        (DirectedGraphQ[T] || Length[VertexList[T]] < 2)
```

The condition on the length of the vertex list is included in the above because *Mathematica* will not recognize a singleton graph or the empty graph as being directed. As promised, this distinguishes between **firstRooted** and its underlying graph.

```
In[14]:= rootedTreeQ[firstRooted]
```

```
Out[14]= True
```

```
In[15]:= rootedTreeQ[UndirectedGraph[firstRooted]]
```

```
Out[15]= False
```

The root of a rooted tree is necessarily the unique vertex with in-degree 0. We can use VertexInDegree to obtain a list of the in-degrees of the vertices of a directed graph.

```
In[16]:= VertexInDegree[firstRooted]
```

```
Out[16]= {0, 1, 1, 1, 1, 1, 1}
```

The Position function, applied to the list of in-degree, will tell us which location in the list is 0. Note that the output of Position is a list of the position specifications, themselves lists.

```
In[17]:= Position[VertexInDegree[firstRooted], 0]
```

```
Out[17]= {{1}}
```

Since TreeGraphQ has identified this as a tree, we know that there will be exactly one vertex with in-degree 0, so we confidently use Part ([[...]]) to obtain the root's position in the list.

```
In[18]:= Position[VertexInDegree[firstRooted], 0][[1, 1]]
```

```
Out[18]= 1
```

Since the output of VertexInDegree matches the order of vertices from VertexList, we can use the previous result with Part ([[...]]) and VertexList to obtain the name of the root.

```
In[19]:= VertexList[firstRooted][[
        Position[VertexInDegree[firstRooted], 0][[1, 1]]
    ]]
```

```
Out[19]= a
```

We create a function based on this example.

```
In[20]:= findRoot[G_?rootedTreeQ] :=
        VertexList[G][[Position[VertexInDegree[G], 0][[1, 1]]]]
```

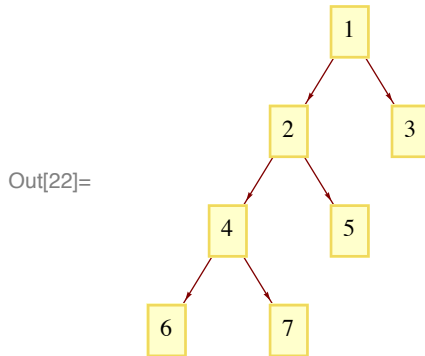
With this function in hand, we can automate the use of TreePlot to draw a rooted tree in its proper

form.

```
In[21]:= rootedPlot[G_?rootedTreeQ, opts___] :=  
         TreePlot[G, Top, findRoot[G], opts]
```

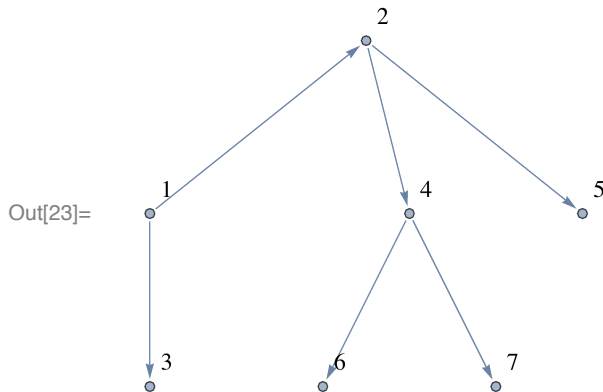
We use this function to draw **secondRooted**.

```
In[22]:= rootedPlot[secondRooted, VertexLabeling → True]
```



The drawback to **rootedPlot** is that, while it produces the expected image of the tree, its output is an image not a Graph object. In particular, **secondRooted** still displays with 2 at the top of the image.

```
In[23]:= secondRooted
```



We can, however, leverage the TreePlot function to modify the Graph object in order to cause it to draw as a rooted tree. To do this, we need to first be able to modify arguments to a function.

Modifying arguments with HoldFirst

We use the HoldFirst attribute to make arguments modifiable. Normally when you provide an argument to a function, the function cannot assign to the argument. Consider the following simple function.

```
In[24]:= five = 5
```

Out[24]= 5

```
In[25]:= addOne[n_] := n = n + 1
```

```
In[26]:= addOne[five]
```

```
Set::setraw : Cannot assign to raw object 5. >>
```

```
Out[26]= 6
```

This function produces an error when we attempt to assign a value to the argument **n**. This is a feature of programming in *Mathematica* that is designed to encourage good programming practices. In particular, for a function to modify one of its arguments, we need to be very explicit that we really want to do so. This helps prevent unintended consequences — accidentally modifying an argument can cause serious bugs in your programs.

You can think about what's going on in the **addOne** function this way: when you call the function with the syntax **addOne[five]**, all of the occurrences of the name **n** are resolved to the object **5**, which is the value stored in **five**. So the command **n=n+1** resolves to **5=5+1**. Clearly that's not a legal command.

The HoldFirst attribute provides a way around this. If we attach the HoldFirst attribute to **addOne**, by applying SetAttributes to the function name and the attribute, we are telling *Mathematica* to not evaluate the parameter **n** into the object it refers to, but to evaluate the argument into a symbol. Instead of evaluating the symbol **five** to get the object **5** and replacing the parameter **n** with **5**, the symbol **five** is held, and the parameter **n** is replaced with the symbol **five**. This means that the command becomes **five=five+1**.

```
In[27]:= SetAttributes[addOne2, HoldFirst];
         addOne2[n_] := n = n + 1
```

Now we can see that this function modifies the variable it is given.

```
In[29]:= addOne2[five]
```

```
Out[29]= 6
```

```
In[30]:= five
```

```
Out[30]= 6
```

To summarize: we can create functions that modify an argument that is passed as a symbol by applying SetAttributes to the function name and the attribute HoldFirst. This allows the parameter to appear on the left side of an assignment and modify the symbol passed as input to the function.

Rooting a Graph Object

Now, in order to draw a Graph object as a rooted tree, we need to better understand the output of TreePlot. Consider the example below.

```
In[31]:= treeplotExample = TreePlot[{1 → 2}, Top, 1]
```

```
Out[31]=
```



We use `FullForm` to expose the underlying data.

```
In[32]:= TreePlot[{1 → 2}, Top, 1] // FullForm
```

```
Out[32]//FullForm=
```

```
Graphics[Annotation[
  GraphicsComplex[List[List[0., 1.], List[0., 0.]],
    List[List[RGBColor[0.5, 0., 0.], Line[List[List[1, 2]]]],
      List[RGBColor[0, 0, 0.7]],
        Tooltip[Point[1], 1], Tooltip[Point[2], 2]]],
  List[], Rule[VertexCoordinateRules,
    List[List[0., 1.], List[0., 0.]]]],
Rule[FrameTicks, None], Rule[PlotRange, All],
Rule[PlotRangePadding, Scaled[0.1]],
Rule[AspectRatio, Automatic]]
```

Notice, in the middle of the output above, the `VertexCoordinateRules` symbol. This is always present in the output from a `TreePlot` and is identified, via a `Rule` (`->`), with a list of vertex locations. Our strategy will be to extract those locations and impose them on the corresponding `Graph`.

To extract the vertex coordinates, we make use of the `ReplaceAll` (`/.`) operator. We typically use `ReplaceAll` (`/.`) with an expression and a rule or a list of rules as below.

```
In[33]:= x^2 + 5 y /. {x → 3, y → 7}
```

```
Out[33]= 44
```

The output from `TreePlot` contains a rule identifying the symbol `VertexCoordinateRules` with the list of vertex locations. But it is not a list of rules, so it cannot be used with `ReplaceAll` (`/.`). The `Cases` function, however, will apply to any expression and produce a list of all the subexpressions matching a pattern. The first argument to `Cases` is an expression, the second is a pattern to search for, and the third optional argument is a level specification. Since the `VertexCoordinateRules` rule is below the top level, we will need to tell `Cases` to delve down to find it. The easiest level specification to use here is `Infinity`. Observe the result of applying `Cases` to our simple example with pattern `_Rule` and level specification `Infinity`.

```
In[34]:= Cases[treeplotExample, _Rule, Infinity]
```

```
Out[34]= {VertexCoordinateRules → {{0., 1.}, {0., 0.}},
  FrameTicks → None, PlotRange → All,
  PlotRangePadding → Scaled[0.1], AspectRatio → Automatic}
```

Using this as the list of rules in conjunction with the ReplaceAll (/.) operator allows us to access the list of vertex locations.

```
In[35]:= VertexCoordinateRules /.
  Cases[treeplotExample, _Rule, Infinity]
```

```
Out[35]= {{0., 1.}, {0., 0.}}
```

Now that we know how to extract the vertex coordinates from the result of TreePlot, we need to be able to set the locations in a Graph object. We do this for an individual vertex by using the PropertyValue function in the form PropertyValue[{*graph*, *vertex*}, VertexCoordinates] = {*x*, *y*}. Our function will loop over all the vertices of the graph and assign the vertex's coordinate to that obtained from the TreePlot.

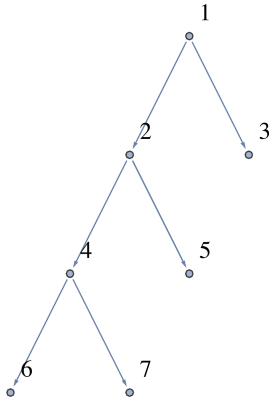
The function is below. It uses HoldFirst in order to modify the graph object it is passed.

```
In[36]:= SetAttributes[drawRooted, HoldFirst];
drawRooted[G_?rootedTreeQ, opts___] :=
Module[{plot, vertList, vertLocs},
  plot = TreePlot[G, Top, findRoot[G], opts];
  vertList = VertexList[G];
  vertLocs =
    VertexCoordinateRules /. Cases[plot, _Rule, Infinity];
  For[i = 1, i ≤ Length[vertList], i++,
    PropertyValue[{G, vertList[[i]]},
      VertexCoordinates] = vertLocs[[i]]
  ];
  G
]
```

Applying this function to **secondRooted** moves the vertices so that it is drawn in the typical places.

```
In[38]:= drawRooted[secondRooted]
```

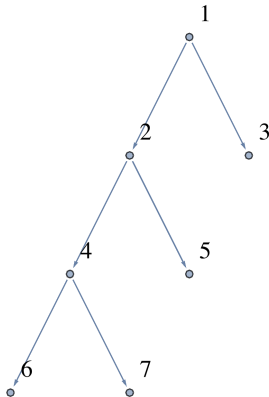
```
Out[38]=
```



Note that this effect is permanent.

```
In[39]:= secondRooted
```

```
Out[39]=
```



Parents, Children, Leaves, and Internal Vertices of Rooted Trees

We now consider functions related to identifying particular vertices in a rooted tree.

We begin with the question of whether one vertex is the parent of another. Given the two vertices, checking this requires determining whether the directed edge from the parent to the child is actually in the tree.

```
In[40]:= parentQ[T_?rootedTreeQ, p_, c_] := EdgeQ[T, DirectedEdge[p, c]]
```

```
In[41]:= parentQ[firstRooted, "b", "f"]
```

```
Out[41]= True
```

```
In[42]:= parentQ[firstRooted, "b", "d"]
```

```
Out[42]= False
```

Next, we consider the question of finding the parent of a given vertex. This can be done by searching the EdgeList output for those edges that have the given vertex at the terminal end. Assuming the graph is in fact a rooted tree, there can be at most one such edge. If the vertex is the root, there will be

no parent and the function will return `Null`. Note the second argument in `Cases` means that it will search the edge list for those edges from the variable p to the given edge v but will put only p in the output.

```
In[43]:= findParent[T_?rootedTreeQ, v_] := Module[{P, p},
  P = Cases[EdgeList[T], DirectedEdge[p_, v] → p];
  If[Length[P] ≠ 1, Null, P[[1]]]
]
```

```
In[44]:= findParent[firstRooted, "d"]
```

```
Out[44]= a
```

```
In[45]:= findParent[firstRooted, "a"]
```

For the related question of determining all children of the given vertex, we take the same approach but with the given vertex at the initial end.

```
In[46]:= findChildren[T_?rootedTreeQ, v_] :=
  Module[{c}, Cases[EdgeList[T], DirectedEdge[v, c_] → c]]
```

```
In[47]:= findChildren[firstRooted, "a"]
```

```
Out[47]= {b, c, d}
```

```
In[48]:= findChildren[firstRooted, "f"]
```

```
Out[48]= {}
```

The `findChildren` function also indicates how we can test a vertex to determine if it is an internal vertex or a leaf.

```
In[49]:= internalVertexQ[T_?rootedTreeQ, v_] :=
  Length[findChildren[T, v]] ≠ 0
```

```
In[50]:= leafQ[T_?rootedTreeQ, v_] := Length[findChildren[T, v]] == 0
```

```
In[51]:= internalVertexQ[firstRooted, "a"]
```

```
Out[51]= True
```

```
In[52]:= leafQ[firstRooted, "a"]
```

```
Out[52]= False
```

```
In[53]:= leafQ[firstRooted, "f"]
```

```
Out[53]= True
```

We can determine all the leaves of a given tree by testing each vertex with `leafQ`. Here we use `Select` in order to test all of the vertices of the graph with a pure `Function` (`&`) based on `leafQ`.

```
In[54]:= findLeaves[T_?rootedTreeQ] := Module[{V},
  V = VertexList[T];
  Select[V, leafQ[T, #] &]
]
```

```
In[55]:= findLeaves[firstRooted]
```

```
Out[55]= {d, e, f, g}
```

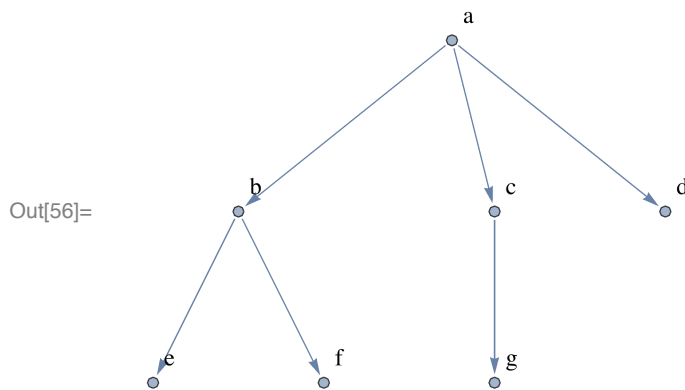
Ordered Rooted Trees

Recall that an ordered rooted tree is a rooted tree in which the children of each internal vertex are ordered.

Representing Ordered Rooted Trees

To represent an ordered rooted tree in *Mathematica*, we'll need to store the order of children. There are many ways to accomplish this, but perhaps the most straightforward is to mark each vertex with its order among its siblings. By way of illustration, we'll make an ordered version of **firstRooted**.

```
In[56]:= firstRooted
```



Mathematica automatically draws the vertices in alphabetical order. But suppose instead we wanted the children of *a* to be in the order *c*, *b*, *d*, and the children of *b* to be in the order *f* then *e*. Internally, we'll represent this by assigning, for each vertex, a property called “order”. We set the order of the root to be 0, and for all other vertices, the order attribute will represent the position of that vertex among its siblings. In our example, *c* will have “order” 1, *b* will have “order” 2, and *d* will have “order” 3. We will define a function that will use PropertyValue to set the “order” property for each vertex.

The first argument to this function will be the name of a rooted tree. The second argument will be a list of values representing the order value to be assigned to each vertex. The function will loop through the vertices of the tree and set the “order” property.

```

In[57]:= setChildOrder::argx =
  "List of orders is of incorrect length.";
setChildOrder[G_?rootedTreeQ, orderL_List] :=
  Module[{T = G, V, v},
    V = VertexList[T];
    Check[
      If[Length[V] ≠ Length[orderL],
        Message[setChildOrder::argx], Return[$Failed]];
      For[i = 1, i ≤ Length[V], i++,
        PropertyValue[{T, V[[i]]}, "order"] = orderL[[i]]
      ];
    T
  ]

```

Since the order values in the list given to the function must match the order of the vertices that is returned from `VertexList`, it is a good idea to double-check the output of `VertexList` before using this function. Also note that this function outputs a new graph, it does not modify the original input, so we will need to make an assignment to the output.

```

In[59]:= VertexList[firstRooted]

```

```

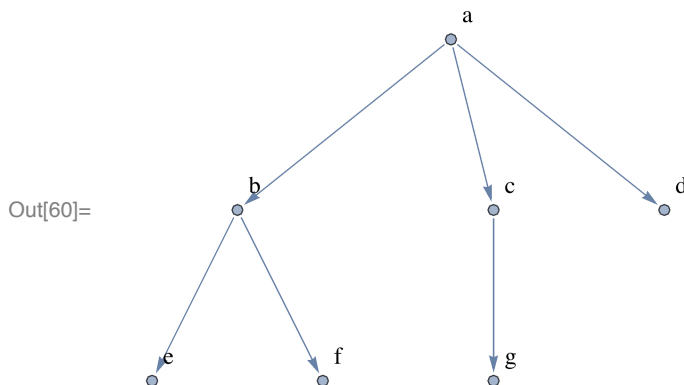
Out[59]= {a, b, c, d, e, f, g}

```

```

In[60]:= firstOrdered = setChildOrder[firstRooted, {0, 2, 1, 3, 2, 1, 1}]

```



Note that the vertices in **firstOrdered** now have the “order” property set, but we still need to make the tree drawn in the appropriate way.

A Test for Ordered Rooted Trees

Now that we know how to represent an ordered rooted tree in *Mathematica*, we will create a function to test a graph object and ensure that it does in fact represent an ordered rooted tree. This will help us make functions we write later be more robust. The requirement for being an ordered rooted tree are: the object must be a tree, it must be directed, every vertex must have the order property set, and the root’s order must be 0.

```

In[61]:= orderedRootedTreeQ[T_] := Module[{v},
  Catch[
    If[! rootedTreeQ, Throw[False]];
    Do[If[
      PropertyValue[{T, v}, "order"] === $Failed, Throw[False]]
      , {v, VertexList[T]}];
    If[PropertyValue[{T, findRoot[T]}, "order"] ≠ 0,
      Throw[False]];
    Throw[True]
  ]
]

```

```

In[62]:= orderedRootedTreeQ[firstRooted]

```

```

Out[62]= False

```

```

In[63]:= orderedRootedTreeQ[firstOrdered]

```

```

Out[63]= True

```

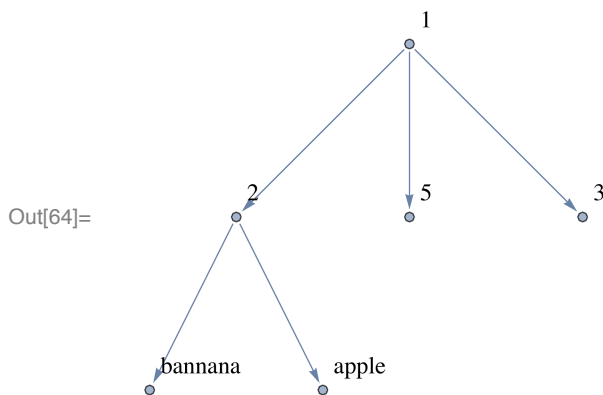
Drawing Ordered Rooted Trees

While **firstOrdered** is now officially an ordered rooted tree and is storing the order of children, it will not be drawn with children in the proper order. To influence the order in which the vertices appear in the drawing, we will use the fact that *Mathematica* draws vertices in the same order as is output by VertexList. Consider the following example,

```

In[64]:= orderExample1 =
  Graph[{1 → 2, 1 → 5, 1 → 3, 2 → "bannana", 2 → "apple"},
    VertexLabels → "Name", ImagePadding → 10]

```



```

In[65]:= VertexList[orderExample1]

```

```

Out[65]= {1, 2, 5, 3, bannana, apple}

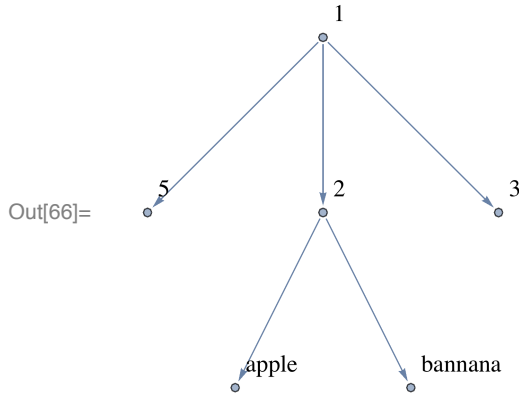
```

The order of the vertex list is determined by the order of the edges. For example, the edge $1 \rightarrow 5$ appears before the edge $1 \rightarrow 3$ in the definition of the Graph, so vertex 5 appears before vertex 3. When drawing the tree, *Mathematica* uses the order of the vertices in the vertex list to determine the relative positions of vertices within a level. In the above, since vertex 5 appears before vertex 3, it also appears

to the left.

By explicitly providing a list of vertices as an argument to `Graph`, we can influence the positions in the image.

```
In[66]:= orderExample2 = Graph[{1, 5, 2, 3, "apple", "bannana"},
    {1 → 2, 1 → 5, 1 → 3, 2 → "bannana", 2 → "apple"},
    VertexLabels → "Name", ImagePadding → 10]
```



We will use this feature in order to properly draw ordered rooted graphs. Given an ordered rooted tree, we'll explicitly sort the vertices based on the order property.

To create the ordered list of vertices, we will begin by initializing a symbol, **OVerts**, to the list containing only the root and we initialize a counter **i** to 1. We then use **findChildren** to find all of the children of the root. We sort the children according to their order property, using **Sort** with a **Function** (&) as the second argument that compares the order properties. Once the children are sorted, they are appended to the **OVerts** list. Then, increment the counter **i** and repeat. At each step, the counter **i** is used as an index into **OVerts** to determine which vertex is being processed. If that vertex has any children, they are sorted and added to the list.

Once the vertex list is properly ordered, we recreate the graph. We use **HoldFirst** again to allow this function to modify its argument.

```

In[67]:= plotORTree[T_?orderedRootedTreeQ, opts___] :=
Module[{E, R, OVerts, i, numverts, children, G, VP, v, p},
  E = EdgeList[T];
  R = findRoot[T];
  OVerts = {R};
  i = 1;
  While[i ≤ Length[OVerts],
    children = findChildren[T, OVerts[[i]]];
    children = Sort[children, PropertyValue[{T, #1}, "order"] ≤
      PropertyValue[{T, #2}, "order"] &];
    OVerts = Join[OVerts, children];
    i++;
  ];
  G = Graph[OVerts, E, opts];
  Do[PropertyValue[{G, v}, "order"] =
    PropertyValue[{T, v}, "order"]
    , {v, OVerts}];
  G
]

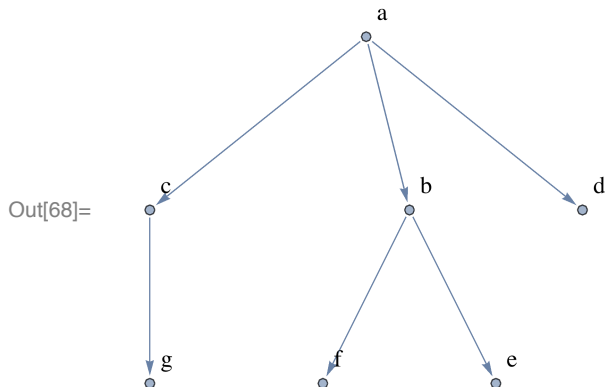
```

Finally, we can draw our ordered tree correctly. Note that to retain the order, we make an explicit reassignment of the symbol.

```

In[68]:= firstOrdered = plotORTree[firstOrdered,
  VertexLabels → "Name", ImagePadding → 10]

```



Properties of Trees

We conclude this section with functions for calculating the level of a vertex, the height of a tree, and for determining if a tree is balanced or not.

The level of a vertex in a rooted tree is the length of the path from the root to the vertex. We compute the level in reverse. We first initialize a counter to 0. If the given vertex is the root, then the level is 0. Otherwise, increment the counter and look at the parent of the original vertex. If this vertex is the root, then the counter holds the level. Otherwise, increment the counter and back up to the parent of the

current vertex. When we reach the root, then the value of the counter is the level of the vertex.

```
In[69]:= findLevel[T_?rootedTreeQ, v_] := Module[{v, level},
  level = 0;
  v = V;
  While[findParent[T, v] != Null,
    level++;
    v = findParent[T, v]
  ];
  level
]
```

We can compute the levels of g and d in the **firstOrdered** tree.

```
In[70]:= findLevel[firstOrdered, "g"]
```

```
Out[70]= 2
```

```
In[71]:= findLevel[firstOrdered, "d"]
```

```
Out[71]= 1
```

The height of a tree is the maximum of the levels of the vertices. We can compute the height by checking each vertex's level. We use a variable to hold the largest level and each time we find a vertex with a level larger than the current maximum, we update the variable.

```
In[72]:= findHeight[T_?rootedTreeQ] := Module[{v, height, level},
  height = 0;
  Do[level = findLevel[T, v];
    If[level > height, height = level]
  , {v, VertexList[T] }];
  height
]
```

```
In[73]:= findHeight[firstOrdered]
```

```
Out[73]= 2
```

Recall that a rooted tree of height h is balanced if all leaves are at level h or $h - 1$. To determine if a given tree is balanced, we need to: (1) calculate the height of the tree, (2) find all the leaves of the tree with the **findLeaves** function we wrote earlier, and (3) test each leaf's level and return **False** if it is higher than level $h - 1$.

```

In[74]:= balancedTreeQ[T_?rootedTreeQ] := Module[{height, leaves, v},
  height = findHeight[T];
  leaves = findLeaves[T];
  Catch[
    Do[If[findLevel[T, v] < height - 1, Throw[False]]
      , {v, leaves]];
  Throw[True]
  ]
]

```

We see that our **firstOrdered** tree is balanced.

```

In[75]:= balancedTreeQ[firstOrdered]
Out[75]= True

```

11.2 Applications of Trees

This section is concerned with applications of trees, particularly binary trees. Specifically, we consider the use of trees in binary search algorithms as well as in Huffman codes. The reason we use binary trees is that we can use the binary structure of the tree to make binary decisions (e.g., less than/greater than) regarding search paths or insertion of elements. Additionally, the binary tree structure corresponds well with the way computers store information as binary data.

Recall that a tree is called a binary tree if all vertices in the tree have at most two children. In this section, we will be using ordered binary trees. The fact that the vertices are ordered means that the children of a vertex can be considered to be either a left child or a right child. By convention, we consider the left child to be the first child and the right child to be second.

Representation in *Mathematica*

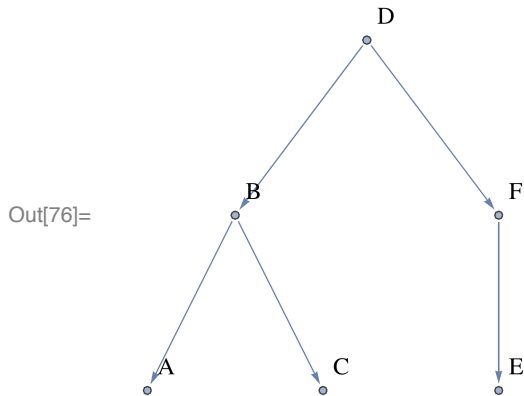
Before we get to the applications, we will discuss how we can represent binary trees in *Mathematica* and develop some functions to help us manipulate them. Since a binary tree is a particular kind of ordered rooted tree, our work here should be consistent with what we did above.

A Binary Tree Test

We will construct a test, **binaryTreeQ** to test whether a graph represents a binary tree. We will impose three conditions for an object to be considered a binary tree. First, it must be an ordered rooted tree, i.e., it must pass **orderedRootedTreeQ**. Second, it must be binary, that is, each vertex must have at most two children. And third, each vertex other than the root must have order attribute 1 or 2, with 1 indicating that the vertex is a left child and 2 for right. The root will have its order attribute set to 0.

First, let's construct an example of a binary tree. The tree we construct is the binary search tree for the letters D, B, F, A, C, E.

```
In[76]:= firstBinaryTree = Graph[{"D", "B", "F", "A", "C", "E"},
  {"D" → "B", "D" → "F", "B" → "A", "B" → "C", "F" → "E"},
  VertexLabels → "Name", ImagePadding → 10]
```



```
In[77]:= firstBinaryTree =
  setChildOrder[firstBinaryTree, {0, 1, 2, 1, 2, 1}];
```

Now that we have an example, let's create the test. To check that the tree is in fact binary, we can use the VertexOutDegree function to count the number of children of each vertex. If any vertex has more than two children, then the tree is not binary. And we'll make sure that each vertex is marked with an order of 1 or 2, or 0 in the case of the vertex.

```
In[78]:= binaryTreeQ[T_Graph] := Module[{R, v, vpos},
  Catch[
    If[! orderedRootedTreeQ[T], Throw[False]];
    R = findRoot[T];
    Do[
      If[VertexOutDegree[T, v] > 2, Throw[False]];
      vpos = PropertyValue[{T, v}, "order"];
      If[(vpos == 0 && v ≠ R) ||
        (vpos ≠ 0 && vpos ≠ 1 && vpos ≠ 2), Throw[False]]
      , {v, VertexList[T]}]
    Throw[True]
  ]
]
```

```
In[79]:= binaryTreeQ[firstBinaryTree]
```

Out[79]= True

Drawing Binary Trees

Next we'll write a function for drawing binary trees. Note that, while **plotORTree** will draw a binary tree, that function will display vertices that are only children directly below their parent rather than to the left or right. Here we take more control over the locations by explicitly calculating the position of each vertex.

Think about the tree as being drawn in a 1 by 1 box with (0, 0) at the bottom left corner. The y-coordinate of each vertex will depend on the height of the tree and the level of the vertex. Specifically, the y coordinate of any vertex is $1 - l/h$, where l is the level of the vertex and h is the height of the tree. This way, the root, which is at level 0, has y-coordinate 1 and the vertices in the last level have y-coordinate 0.

For the x-coordinates, the position of the vertex depends on the position of its parent and its level. We start by setting the x-position of the root to $\frac{1}{2}$. The left child of the root will be drawn at x-coordinate $\frac{1}{4}$ and the right child at $\frac{3}{4}$. We can think about the children of the root as being drawn $\frac{1}{4}$ to the left of the root and $\frac{1}{4}$ to the right of the root, respectively. That is, the x-coordinate of the root's left child is $\frac{1}{2} - \frac{1}{4}$, and the x-coordinate of the right child is $\frac{1}{2} + \frac{1}{4}$. Generally, for a vertex in level l , we can calculate its x-coordinate as the x-coordinate of its parent plus (for right children) or minus (for left) $\frac{1}{2^{l+1}}$.

The **drawBinaryTree** function is below. It begins by calculating the height of the tree with the **findHeight** function we created earlier and finds the root with **findRoot**. It then processes the root of the tree by setting its position to $(\frac{1}{2}, 1)$. (Note that we can set the position of the vertex by using PropertyValue to set the VertexCoordinates property. This is in some ways more convenient than assembling all of the vertex positions and then using VertexCoordinates within a Graph definition.) We also create a list, **verts**, and populate it with the root's children.

We then begin a While loop. We initialize a counter **i** to 1. This counter serves as an index into the **verts** list. At each step in the loop, we do several things. First, we use the **findLevel** function to determine the level of the vertex. Second, the y-coordinate is calculated by the formula $1 + \frac{l}{h}$. Third, the x-coordinate is calculated by accessing the x-coordinate of the parent and adding or subtracting $\frac{1}{2^{l+1}}$. Fourth, we set the VertexCoordinates property for the vertex. And finally, the children of the current vertex are added to the **verts** list and the counter is incremented.

```

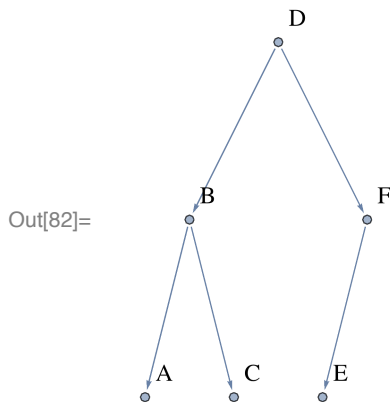
In[80]:= SetAttributes[drawBinaryTree, HoldFirst];
drawBinaryTree[T_?binaryTreeQ] :=
Module[{height, v, i, level, verts, x, y, parent, side},
  height = findHeight[T];
  v = findRoot[T];
  PropertyValue[{T, v}, VertexCoordinates] = {1/2, 1};
  verts = findChildren[T, v];
  i = 1;
  While[i ≤ Length[verts],
    v = verts[[i]];
    level = findLevel[T, v];
    y = 1 - level / height;
    parent = findParent[T, v];
    x = PropertyValue[{T, parent}, VertexCoordinates] [[1]];
    side = Switch[PropertyValue[{T, v}, "order"], 1, -1, 2, 1];
    x = x + side * 1 / (2 ^ (level + 1));
    PropertyValue[{T, v}, VertexCoordinates] = {x, y};
    verts = Join[verts, findChildren[T, v]];
    i++;
  ];
  T
]

```

```

In[82]:= drawBinaryTree[firstBinaryTree]

```



Parents and Children

In the previous section, we created the function **findParent**, which returns the parent of a given vertex in the given tree. This function works on binary trees just as well.

```

In[83]:= findParent[firstBinaryTree, "C"]

```

Out[83]= B

We had also created the **findChildren** function, which we can also use with binary trees. But for binary trees, we'll want to be more specific and be able to determine the left and right children of a

given vertex. Finding the left (respectively, right) child of a given vertex can be done by looking at each child of the vertex and checking the order attribute. The child with order 1 is the left child and is returned by the function (respectively, 2 and right child). If there is no left (respectively, right), child, the function will return Null.

```
In[84]:= findLeftChild[T_?binaryTreeQ, v_] /; VertexQ[T, v] :=
  Module[{children, w, pos},
    children = findChildren[T, v];
    Catch[
      Do[pos = PropertyValue[{T, w}, "order"];
        If[pos == 1, Throw[w]]
        , {w, children}];
      Throw[Null]
    ]
  ]

In[85]:= findRightChild[T_?binaryTreeQ, v_] /; VertexQ[T, v] :=
  Module[{children, w, pos},
    children = findChildren[T, v];
    Catch[
      Do[pos = PropertyValue[{T, w}, "order"];
        If[pos == 2, Throw[w]]
        , {w, children}];
      Throw[Null]
    ]
  ]

In[86]:= findLeftChild[firstBinaryTree, "F"]

Out[86]= E

In[87]:= findRightChild[firstBinaryTree, "F"]
```

Building Binary Trees

We will also want functions to create and build up a binary tree. Specifically, we'll create a function that, given the label for the root of a binary tree, creates the tree with that vertex as its root. We will then write functions that, given a binary tree, a vertex in the tree, and a label for a new vertex, adds the new vertex as the left or right child of the given vertex.

The **newBinaryTree** function creates the binary tree consisting of a single vertex, the root of the tree. This function also accepts a list of options to be passed to the Graph.

```
In[88]:= newBinaryTree[r_, opts___] := Module[{T},
  T = Graph[{r}, {}, opts];
  PropertyValue[{T, r}, "order"] = 0;
  drawBinaryTree[T]
]
```

Adding a child to a vertex in a binary tree requires three basic steps. We must add a vertex to the graph with the VertexAdd function, add a directed edge from the parent to the new vertex with the

EdgeAdd function, and then use PropertyValue to identify the new child as left or right by setting the order attribute to 1 or 2, respectively. We conclude the functions with a call to **drawBinaryTree** so as to ensure the resulting tree appears with the vertices in the correct order.

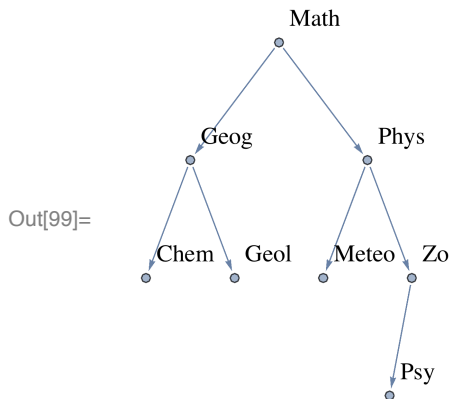
```
In[89]:= addLeftChild[T_?binaryTreeQ, v_, newV_] /; VertexQ[T, v] :=
Module[{newT, newedge, temp, w},
  If[findLeftChild[T, v] ≠ Null, Return[$Failed]];
  newT = VertexAdd[T, newV];
  newT = EdgeAdd[newT, DirectedEdge[v, newV]];
  PropertyValue[{newT, newV}, "order"] = 1;
  drawBinaryTree[newT]
]

In[90]:= addRightChild[T_?binaryTreeQ, v_, newV_] /; VertexQ[T, v] :=
Module[{newT, newedge, temp, w},
  If[findRightChild[T, v] ≠ Null, Return[$Failed]];
  newT = VertexAdd[T, newV];
  newT = EdgeAdd[newT, DirectedEdge[v, newV]];
  PropertyValue[{newT, newV}, "order"] = 2;
  drawBinaryTree[newT]
]
```

With **newBinaryTree**, **addLeftChild**, and **addRightChild**, we can now construct binary trees one vertex at a time. We will illustrate this by creating the binary search tree described in Example 1 of the text by following the steps illustrated in Figure 1. We abbreviate the words in order to make the image more readable.

```
In[91]:= fig1Tree = newBinaryTree["Math",
  VertexLabels → "Name", ImagePadding → 10];
fig1Tree = addRightChild[fig1Tree, "Math", "Phys"];
fig1Tree = addLeftChild[fig1Tree, "Math", "Geog"];
fig1Tree = addRightChild[fig1Tree, "Phys", "Zoo"];
fig1Tree = addLeftChild[fig1Tree, "Phys", "Meteo"];
fig1Tree = addRightChild[fig1Tree, "Geog", "Geol"];
fig1Tree = addLeftChild[fig1Tree, "Zoo", "Psy"];
fig1Tree = addLeftChild[fig1Tree, "Geog", "Chem"];
```

In[99]:= **fig1Tree**



Binary Insertion

A key benefit of binary search trees is that the search time required to find a specific element of the tree is logarithmic in the number of vertices of the tree. The drawback is that the initial insertion of a vertex is more expensive.

The procedure for constructing a binary search tree by insertion is described in Algorithm 1 in Section 11.2 of the textbook. We will implement this algorithm as the function **binaryInsertion**.

The **binaryInsertion** function will accept two input values: a binary search tree and a vertex to be found or added. The function returns True if the vertex is found to already be in the tree, and if not, it will add the vertex to the tree and return False.

We begin by locating the root of the tree and setting the local variable **v** to the root. Then we begin a While loop. This loop continues provided two conditions are met. First, that **v** is not Null. If we discover that the value we're searching for is not in the tree, then we will add it to the tree and set **v** to Null, to indicate that we had to add a vertex, and this terminates the While loop. The second condition is that **v** is not **x**, where **x** is the value we are searching for. If **v = x**, then we have found the vertex and thus the loop should terminate. (Note that *Mathematica* identifies a vertex with its label, so, unlike the text, we do not distinguish between **v** and its label.)

Within the While loop, there are two possibilities. Either the sought-after value is less than **v** or it is greater than **v**. They cannot be equal because that is one of the terminating conditions for the While loop. If the target value is less than **v**, then we consider the left child of **v**. If there is no left child, then we know that the value is not in the tree and so we add the value as the left child of **v** and then set **v** to Null to indicate that the desired value was not already in the tree. If there is a left child, then we set **v** equal to it and continue the loop. If the target value is greater than **v**, we proceed in exactly the same way, substituting right for left.

Once the loop terminates, we check the value of **v**. If **v** is Null, then we know that the desired value was not found and the algorithm returns False. If **v** is not Null, then we return True.

Here, now, is the binary insertion algorithm.

```

In[100]:= SetAttributes[binaryInsertion, HoldFirst];
binaryInsertion[BST_?binaryTreeQ, x_] := Module[{v},
  v = findRoot[BST];
  While[v != Null && v != x,
    If[Order[x, v] == 1,
      If[findLeftChild[BST, v] == Null,
        (* no left child *)
        BST = addLeftChild[BST, v, x];
        v = Null,
        (* left child exists *)
        v = findLeftChild[BST, v]]
      , (* else x>v *)
      If[findRightChild[BST, v] == Null,
        (* no right child *)
        BST = addRightChild[BST, v, x];
        v = Null,
        (* right child exists *)
        v = findRightChild[BST, v]]
    ]
  ]; (* end while loop *)
  v != Null
]

```

Note that we use the Order function to compare **x** and **v**. Order returns 1 if the first argument is before (“less than”) the second in the canonical order, i.e., alphabetically for strings, numerically for numbers. *Mathematica* does not allow you to use the less-than operator, Less (<) with strings. Order is generic and will allow both string and integer vertex names.

Also note that **binaryInsertion** requires that its argument already be a binary tree. A new tree must be created with **newBinaryTree**.

Now let's see if oceanography is in the **fig1Tree** of academic subjects.

```

In[102]:= binaryInsertion[fig1Tree, "Oc"]

```

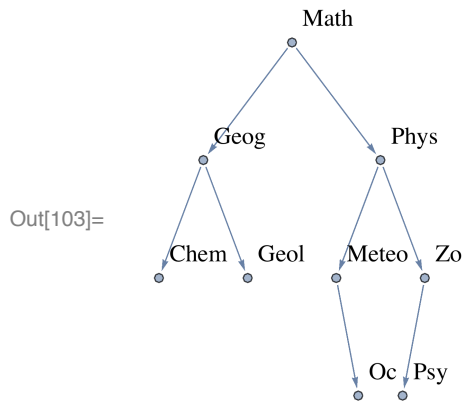
```

Out[102]= False

```

The function returned False indicating that “Oc” was not found in the tree. Displaying **fig1Tree**, we see that it was added as a child of meteorology.

In[103]:= **fig1Tree**

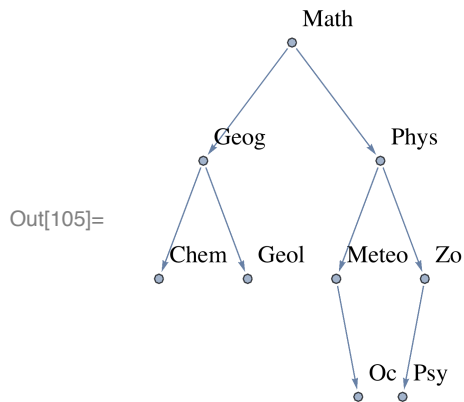


On the other hand, zoology is already in the tree and so the tree is not modified.

In[104]:= **binaryInsertion[fig1Tree, "Zoo"]**

Out[104]= **True**

In[105]:= **fig1Tree**



Constructing a Binary Search Tree from a List

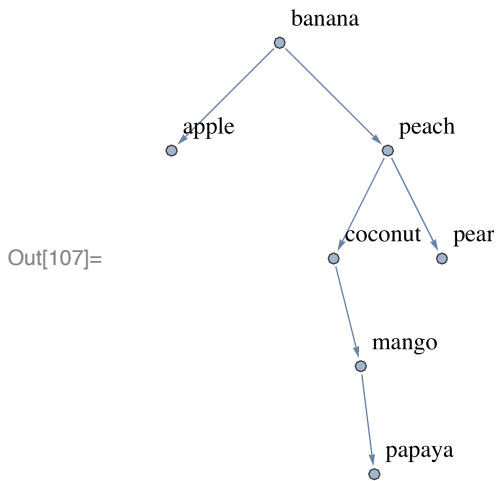
To conclude our discussion of binary search trees, we will create a function that takes a list of values and successively uses the **binaryInsertion** function to create a binary search tree for the given list.

```

In[106]:= makeBST[L_List, opts___] := Module[{T, v},
  T = newBinaryTree[L[[1]], opts];
  Do[binaryInsertion[T, L[[i]]], {i, 2, Length[L]}];
  T
]
```

We use this to complete Exercise 1 from Section 11.2.

```
In[107]:= exercisel =
makeBST[{"banana", "peach", "apple", "pear", "coconut",
          "mango", "papaya"}, VertexLabels → "Name", ImagePadding → 15]
```



Huffman Coding

Huffman coding is a method for constructing an efficient prefix code for a set of characters. It is based on a greedy algorithm, where at each step the vertices with the least weight are combined. It can be shown that Huffman coding produces optimal prefix codes. The algorithm that we will implement is described in Algorithm 2 of Section 11.2.

Creating the Initial Forest

We begin with a list of symbols and their weights, or frequencies. The first step is to create the initial forest, which we will implement as a list of trees. For each symbol, we will create the binary tree consisting of a single vertex corresponding to the symbol.

We create a function, similar to **newBinaryTree**, which, in addition to creating the binary tree, also assigns a “weight” attribute to the graph to store the weight of the symbol.

```
In[108]:= newHTree[s_, w_, opts___] := Module[{T},
  T = Graph[{s}, {}, opts];
  PropertyValue[T, "weight"] = w;
  PropertyValue[{T, s}, "order"] = 0;
  T
]
```

Now we write a function to create the initial forest. We assume that the data is provided as a list of pairs consisting of the symbol and the weight.

```

In[109]:= createForest[L_List, opts___] :=
  Module[{forest = {}, M, v, w, G},
    Do[{v, w} = M;
      AppendTo[forest, newHTree[v, w, opts]]
    , {M, L}];
    forest
  ]

```

Using this function, we form the initial forest for Exercise 23 from Section 11.2. We use the ImageSize option to limit the width of the images. Ordinarily, you would simply suppress the output from this function.

```

In[110]:= ex23Forest = createForest [
  {"a", 0.20}, {"b", 0.10}, {"c", 0.15}, {"d", 0.25},
  {"e", 0.30}], VertexLabels → "Name", ImageSize → 50]

```

```

Out[110]= {
  •a ,      •b ,      •c ,      •d ,      •e }

```

The main work of the Huffman coding algorithm is to determine the two members of the forest with the smallest weights. These two trees are then assembled into a single tree whose root is a new vertex and with the trees with the lowest weight and second lowest weight as the right and left subtrees of the root. The new tree's weight is the sum of the weights of the two original trees.

Sorting the Forest

Recall that the Sort function accepts an optional second argument, specifically, a function on two arguments that returns True if the first argument precedes the second in the desired order. The following function will be of this kind, accepting two binary trees as input and comparing their weights.

```

In[111]:= compareTrees[A_?binaryTreeQ, B_?binaryTreeQ] := Module[{a, b},
  a = PropertyValue[A, "weight"];
  b = PropertyValue[B, "weight"];
  a < b
]

```

```

In[112]:= ex23Forest = Sort[ex23Forest, compareTrees]

```

```

Out[112]= {
  •b ,      •c ,      •a ,      •d ,      •e }

```

Combining Two Trees

Next, we need to take two binary trees and create a new binary tree with one tree as the left subtree of the new root and the other as the right subtree. This function will require three arguments: the name of the new root, the left subtree, and the right subtree.

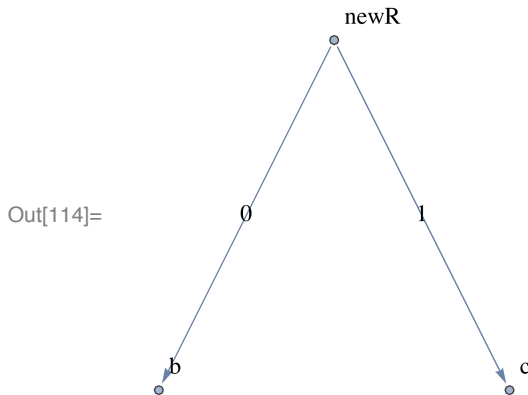
We create the new tree by: (1) combining the vertex lists of the original trees and adding the new vertex; (2) merging the two sets of edges of the original trees and adding two new edges linking the

new root to the previous roots; (3) copying the “position” attribute from the original trees and then changing the position attributes for the two original roots to reflect their new status as left and right children in the new tree; (4) copying the edge labels and assigning the new edges the labels 0 and 1; and (5) setting the “weight” attribute to be the sum of the weights of the two original trees.

```
In[113]:= joinHTrees[newR_, A_?binaryTreeQ, B_?binaryTreeQ, opts___] :=
Module[{newT, newVerts, Aroot, Broot, newEdges, v, e, p, w},
  newVerts = Join[{newR}, VertexList[A], VertexList[B]];
  Aroot = findRoot[A];
  Broot = findRoot[B];
  newEdges = Join[EdgeList[A], EdgeList[B],
    {DirectedEdge[newR, Aroot], DirectedEdge[newR, Broot]}];
  newT = Graph[newVerts, newEdges, opts];
  Do[PropertyValue[{newT, v}, "order"] =
    PropertyValue[{A, v}, "order"]
    , {v, VertexList[A]}];
  Do[PropertyValue[{newT, v}, "order"] =
    PropertyValue[{B, v}, "order"]
    , {v, VertexList[B]}];
  PropertyValue[{newT, Aroot}, "order"] = 1;
  PropertyValue[{newT, Broot}, "order"] = 2;
  PropertyValue[{newT, newR}, "order"] = 0;
  Do[PropertyValue[{newT, e}, EdgeLabels] =
    PropertyValue[{A, e}, EdgeLabels]
    , {e, EdgeList[A]}];
  Do[PropertyValue[{newT, e}, EdgeLabels] =
    PropertyValue[{B, e}, EdgeLabels]
    , {e, EdgeList[B]}];
  PropertyValue[
    {newT, DirectedEdge[newR, Aroot]}, EdgeLabels] = 0;
  PropertyValue[{newT, DirectedEdge[newR, Broot]},
    EdgeLabels] = 1;
  w = PropertyValue[A, "weight"] + PropertyValue[B, "weight"];
  PropertyValue[newT, "weight"] = w;
  newT
]
```

For example, we'll join the first two graphs in our sorted **ex23Forest**.

```
In[114]:= exampleJoin = joinHTrees["newR", ex23Forest[[1]],
    ex23Forest[[2]], VertexLabels -> "Name", ImagePadding -> 10]
```



Implementing the Main Function

We now have the major pieces of the Huffman algorithm assembled and we can write the **huffmanCode** function. This function will accept as input the same list of symbol-weight pairs as the **createForest** function did. The function's first step is to create the forest **F**. Then we begin a While loop that continues as long as the list **F** contains more than one element. Inside the loop, we first use the **compareTrees** function to sort the forest in increasing order of weight. Then, we use the **joinHTrees** function to join the first two trees in the forest and we add that new tree to the list, replacing the original two.

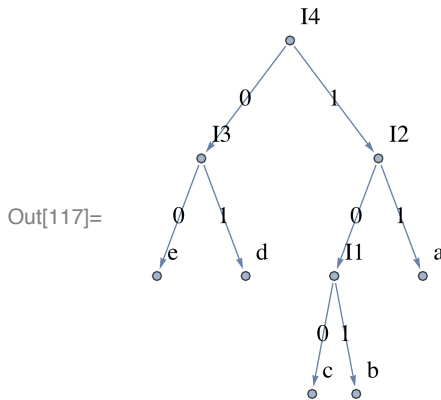
```
In[115]:= huffmanCode[L_List, opts___] := Module[{F, i, tempT},
    F = createForest[L, opts];
    i = 0;
    While[Length[F] > 1,
        F = Sort[F, compareTrees];
        i++;
        tempT = joinHTrees["I" <> ToString[i], F[[2]], F[[1]], opts];
        F = Append[F[[3 ;; -1]], tempT]
    ];
    F[[1]]
]
```

Note that we need a name for the new root when we join two trees. Since these will be the internal vertices of the final tree, we'll call them I1, I2, I3, etc. We keep a counter **i** and use the string concatenation operator StringJoin (<>) to create the names of the internal vertices.

```

In[116]:= ex23HCode = huffmanCode[{{"a", 0.20},
    {"b", 0.10}, {"c", 0.15}, {"d", 0.25}, {"e", 0.30}},
    VertexLabels → "Name", ImagePadding → 10];
drawBinaryTree[ex23HCode]

```



Encoding Strings Using the Huffman Code Tree

We conclude this section by writing a function to encode a string of symbols using a given Huffman tree. Note that you can use `Characters` to decompose a string into a list of the individual letters. For example,

```

In[118]:= Characters["Hello"]

```

```

Out[118]= {H, e, l, l, o}

```

Since we encode a string by assembling the codes for individual characters, we'll start with a function for encoding a single character. We assemble the character's code right to left. Beginning with the vertex corresponding to the desired character, we find that vertex's parent. The last digit of the code is the weight of the corresponding edge. The next rightmost digit is the weight of the edge connecting the next parent. We continue until we reach the root. We use the `StringLength` function in this function to determine the length of the input string and ensure it contains only a single character.

```

In[119]:= encodeCharacter[H_?binaryTreeQ, c_String] /;
    StringLength[c] == 1 := Module[{code, vertex, parent, digit},
    vertex = c;
    code = "";
    While[findParent[H, vertex] != Null,
        parent = findParent[H, vertex];
        digit = PropertyValue[
            {H, DirectedEdge[parent, vertex]}, EdgeLabels];
        code = ToString[digit] <> code;
        vertex = parent
    ];
    code
]

```

```
In[120]:= encodeCharacter[ex23HCode, "c"]
```

```
Out[120]= 100
```

To encode a string, we encode each character and assemble the results.

```
In[121]:= encodeString[H_?binaryTreeQ, S_String] :=
Module[{charList, code, char, charcode},
  charList = Characters[S];
  code = "";
  Do[
    charcode = encodeCharacter[H, char];
    code = code <> charcode
    , {char, charList}];
  code
]
```

We use this to encode the word “ace”.

```
In[122]:= encodeString[ex23HCode, "ace"]
```

```
Out[122]= 1110000
```

11.3 Tree Traversal

In this section we show how to use *Mathematica* to carry out tree traversals. Recall that a tree traversal algorithm is a procedure for systematically visiting every vertex of an ordered rooted tree. We will provide procedures for three important tree traversal algorithms: preorder traversal, inorder traversal, and postorder traversal. We will then show how to use these traversal methods to produce the prefix, infix, and postfix notations for arithmetic expressions.

These tree traversal algorithms all require that the tree be rooted and ordered. Recall how we implemented ordered rooted trees in Section 1. A tree that satisfies **orderedRootedTree** is a Graph that is a rooted tree and with the additional restriction that each vertex has an “order” attribute.

Also recall that we illustrated in Section 1 how to use Sort to sort lists of vertices based on the “order” attribute. This is done by giving Sort a second argument, which is a pure Function (&) that accesses and compares the “order” attribute of two vertices of the tree.

To begin, we will create an ordered tree to use as an example as we explore the three traversal algorithms. This example is a reproduction of Figure 3 from Section 11.3.

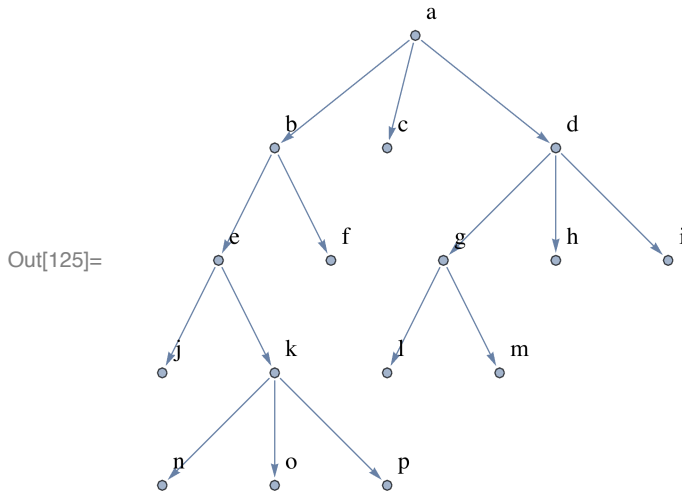
```
In[123]:= fig3Tree =
  Graph[{ "a" → "b", "a" → "c", "a" → "d", "b" → "e", "b" → "f",
    "d" → "g", "d" → "h", "d" → "i", "e" → "j", "e" → "k",
    "g" → "l", "g" → "m", "k" → "n", "k" → "o", "k" → "p"},
  VertexLabels → "Name", ImagePadding → 10];
```

```
In[124]:= VertexList[fig3Tree]
```

```
Out[124]= {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p}
```

```
In[125]:= fig3Tree = setChildOrder[fig3Tree,  

           {0, 1, 2, 3, 1, 2, 1, 2, 3, 1, 2, 1, 2, 1, 2, 3}];
```



Subtrees

Before implementing the traversal algorithms, we need a function that determines a subtree of a tree. In particular, we want an function that, given a tree and a vertex, will return the subtree with the given vertex as the root and that includes all of its descendants.

To produce the subtree, we will use the Subgraph function and then copy the “order” attributes to the new tree from the original. The vertices that we want included in the subgraph are the given vertex, which will be the root of the subtree, together with all of its descendants. We begin by creating a function that finds all of the descendants of the given vertex. This function can apply to any rooted tree.

The approach is the same as we've used before. We begin with the given vertex and create a list consisting of its children, which we obtain with the **findChildren** function. We then begin a loop over the list of descendants. At each step, we add all of the children of the current vertex to the list, and then move on to the next vertex in the list. This continues until we reach the end of the list and there are no more children to add. (Note: this is referred to as a level-order traversal.)

```
In[126]:= descendants[T_?rootedTreeQ, parent_] := Module[{dList, v, i},  

           dList = findChildren[T, parent];  

           i = 1;  

           While[i ≤ Length[dList],  

             v = dList[[i];  

             dList = Join[dList, findChildren[T, v]];  

             i++  

           ];  

           dList  

]
```

Compute the descendants of *e* in the example tree above.

```
In[127]:= descendants[fig3Tree, "e"]
```

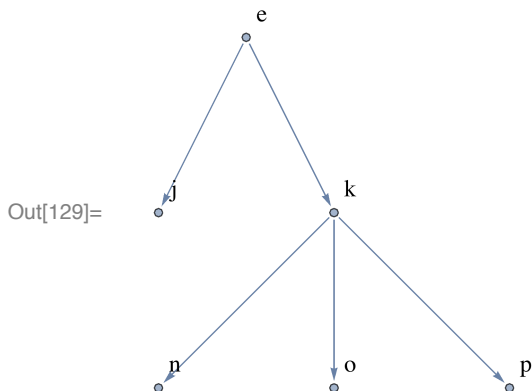
```
Out[127]= {j, k, n, o, p}
```

To construct the subtree of an ordered rooted tree with a given vertex as its root, we need to: find the descendants of the given vertex; use the `Subgraph` function on the given vertex and all its descendants; copy the “order” attribute to the new tree; and set the “order” attribute of the root to 0.

```
In[128]:= subTree[T_?orderedRootedTreeQ, newRoot_, opts___] :=
Module[{vList, subT, v},
  vList = descendants[T, newRoot];
  PrependTo[vList, newRoot];
  subT = Subgraph[T, vList, opts];
  Do[PropertyValue[{subT, v}, "order"] =
    PropertyValue[{T, v}, "order"]
    , {v, vList}];
  PropertyValue[{subT, newRoot}, "order"] = 0;
  subT
]
```

Let's check this function by finding the subtree with root *e* and making sure it satisfies `orderedRootedTreeQ`.

```
In[129]:= subExample = subTree[fig3Tree, "e",
  VertexLabels → "Name", ImagePadding → 10]
```



```
In[130]:= orderedRootedTreeQ[subExample]
```

```
Out[130]= True
```

Traversal Algorithms

We now implement the three traversal algorithms described in Section 11.3 of the text. We begin with the preorder traversal algorithm, which is given as Algorithm 1 in the text.

Preorder

Given an ordered rooted tree, the preorder algorithm acts as follows. First, it prints the name of the root. Then it calculates the children of the root and stores them in order. For each child, in order, the

function recursively applies itself to the subtree with the given child as root.

```
In[131]:= preorder[T_?orderedRootedTreeQ] :=
Module[{root, children, i, tempSubT},
  root = findRoot[T];
  Print[root];
  children = findChildren[T, root];
  children = Sort[children, (PropertyValue[{T, #1}, "order"] <
    PropertyValue[{T, #2}, "order"])] &];
  For[i = 1, i ≤ Length[children], i++,
    tempSubT = subTree[T, children[[i]]];
    preorder[tempSubT];
  ]
]
```

```
In[132]:= preorder[fig3Tree]
```

```
a
b
e
j
k
n
o
p
f
c
d
g
l
m
h
i
```

You can confirm that this output is consistent with the preorder traversal demonstrated in Figure 4 of Section 11.3 of the textbook.

Postorder

Postorder traversal, described in Algorithm 3 of the text, is very similar to preorder traversal. The only change needed in the code is that, instead of printing the root at the start of the algorithm, the vertex is printed after the loop is completed.

```

In[133]:= postorder[T_?orderedRootedTreeQ] :=
  Module[{root, children, i, tempSubT},
    root = findRoot[T];
    children = findChildren[T, root];
    children = Sort[children, (PropertyValue[{T, #1}, "order"] <
      PropertyValue[{T, #2}, "order"]) &];
    For[i = 1, i ≤ Length[children], i++,
      tempSubT = subTree[T, children[[i]]];
      postorder[tempSubT];
    ];
    Print[root]
  ]

```

```

In[134]:= postorder[fig3Tree]

```

```

j
n
o
p
k
e
f
b
c
l
m
g
h
i
d
a

```

Inorder

In inorder traversal, the algorithm first applies itself recursively to the first child of the vertex, then it prints the vertex, and then it applies itself to the remainder of the children, in order.

```

In[135]:= inorder[T_?orderedRootedTreeQ] :=
  Module[{root, children, i, tempSubT},
    root = findRoot[T];
    children = findChildren[T, root];
    children = Sort[children, (PropertyValue[{T, #1}, "order"] <
      PropertyValue[{T, #2}, "order"])] &];
    If[Length[children] ≠ 0,
      tempSubT = subTree[T, children[[1]]];
      inorder[tempSubT]
    ];
    Print[root];
    For[i = 2, i ≤ Length[children], i++,
      tempSubT = subTree[T, children[[i]]];
      inorder[tempSubT];
    ]
  ]

In[136]:= inorder[fig3Tree]

j
e
n
k
o
p
b
f
a
c
l
g
m
d
h
i

```

Infix Notation

In the remainder of this section, we discuss how to use *Mathematica* to work with the infix, prefix, and postfix forms of arithmetic expressions, as described in Section 11.3 of the text. In this subsection, we

will show how to create a tree representation of an infix expression. In the next subsection, we will explore how to evaluate expressions from their postfix and prefix forms.

Recall that infix notation is the usual notation for basic arithmetic and algebraic expressions. We will construct a *Mathematica* function that takes an infix expression and converts it into a tree representation. This tree representation can then be traversed using the traversals of the previous sections to form various arithmetic representation formats.

The algorithm we use to turn an arithmetic expression in infix notation into a tree is recursive. The basis case occurs when the expression consists of a single number or variable. In this case, the tree consists of a single vertex.

Otherwise, the expression consists of a left operand, an operator, and a right operand. In this case, we (1) apply the algorithm to the left and right operands, and (2) combine the resulting trees with the operator as the common root. Implementing this will require some preliminary work.

First, we need to represent arithmetic expressions in *Mathematica* in such a way that we can work with them and ensure that *Mathematica* won't evaluate them. This can be accomplished by applying `Hold` to the expression when entering it, as illustrated below.

```
In[137]:= Hold[2 + 3 * 5]
```

```
Out[137]= Hold[2 + 3 * 5]
```

Second, we need to be able to distinguish the basis case from the recursive case.

Third, the leaves in the tree will be numbers and variables and the internal vertices will be operations. In an expression like $3 \cdot 4 + 7 \cdot (3 + x)$, we have repetition among the operators and the operands (two 3's, two additions, and two multiplications). We need a way to get *Mathematica* to consider each of these to be a distinct object, since *Mathematica* insists that the vertices in a graph be distinct, while displaying them with the same symbol in the tree.

Fourth, in the recursive step, we need to be able to identify the operator and separate the left and right operands.

And fifth, we will need to implement a function to perform the combination of subtrees described in part (2) of the recursive step.

Distinguishing the Basis and Recursive Cases

Any arithmetic expression is either a single integer or variable, or it is two expressions joined by an arithmetic operator.

We can determine which kind it is by testing the head of the expression.

```
In[138]:= Head[5]
```

```
Out[138]= Integer
```

```
In[139]:= Head[x]
```

```
Out[139]= Symbol
```

However, observe that an expression such as $3 + 5$ also reports having head `Integer`.

```
In[140]:= Head[3 + 5]
```

```
Out[140]= Integer
```

This is because *Mathematica* evaluates the arithmetic before apply the Head function. Recall from above that we will be using Hold to prevent arithmetic evaluation and algebraic simplification. But with Hold in place, Head will identify the head of the expression as Hold.

```
In[141]:= Head[Hold[3 + 5]]
```

```
Out[141]= Hold
```

To obtain the correct head of an expression such as $3 + 5$ while preventing evaluation, we need to use Part (`[[...]]`). Remember that 0 represents the head within a Part (`[[...]]`).

```
In[142]:= 5[[0]]
```

```
Out[142]= Integer
```

For a held expression, `[[1, 0]]` will give the head of the expression within the Hold.

```
In[143]:= Hold[3 + 5][[1, 0]]
```

```
Out[143]= Plus
```

Ensuring That Each Occurrence of an Object Is Considered Distinct

The tree associated to the expression $3 \cdot 4 + 7 \cdot (3 + x)$ will have 9 vertices. The internal vertices, the operations, consist of two additions and two multiplications. The leaves, the numbers and variables, consist of 4, 7, x , and two 3s.

In *Mathematica* graphs, each vertex must be unique and distinct from all other vertices. In order to make *Mathematica* consider two 3s or two additions to be different, we will take advantage of the fact that a Module structure attaches an integer to each variable name in order to ensure it is unique. The following reveals the internal symbol associated to a local variable.

```
In[144]:= Module[{var}, ToString[var]]
```

```
Out[144]= var$2829
```

To use this while building our expression tree, we will define an alternate version of **newBinaryTree**, which was defined earlier in the chapter. This new function, **newExpressionTree**, will declare a local variable and set it equal to the result of applying ToString to the variable. Since the local variable had not previously been assigned, this will store the internal representation of the variable. We then use that as the name of the vertex in a new tree. We set the VertexLabels property in order to have the graph displayed with the elements of the expression rather than the internal name.

```
In[145]:= newExpressionTree[r_, opts___] := Module[{T, v},
  v = ToString[v];
  T = Graph[{v}, {}, opts];
  PropertyValue[{T, v}, VertexLabels] = r;
  PropertyValue[{T, v}, "order"] = 0;
  drawBinaryTree[T]
]
```

Now we are able to create trees and we will be assured that each one will have distinct vertex names, internally.

Identifying the Operator and Operands

In the recursive case, we must separate a complex expression into its operator and the left and right operands. Consider the following example.

```
In[146]:= exampleExpr = Hold[3 * 4 + 7 * (3 + x)]
```

```
Out[146]= Hold[3 × 4 + 7 (3 + x)]
```

This expression, $3 \cdot 4 + 7 \cdot (3 + x)$ consists of the sum of $3 \cdot 4$ and $7 \cdot (3 + x)$. In *Mathematica*, we will obtain these two operands by applying Extract.

Extract is similar to Part ([[...]]) or Take in that it returns a part of an expression. However, it allows an optional third argument to apply a head to the part of the expression before it is returned. This is important here because we need to prevent *Mathematica* from simplifying the operands.

The arguments to Extract will be the expression, the part specification, and the head Hold. To see what the part specification should be, look at the full form of **exampleExpr**.

```
In[147]:= FullForm[exampleExpr]
```

```
Out[147]//FullForm=
```

```
Hold[Plus[Times[3, 4], Times[7, Plus[3, x]]]]
```

We see that the expression is entirely enclosed in Hold, so part specification **{1}** will refer to the entire algebraic expression, which has head Plus. The arguments of the top Plus are then referenced as **{1,1}** and **{1,2}**.

```
In[148]:= Extract[exampleExpr, {1, 1}, Hold]
```

```
Out[148]= Hold[3 × 4]
```

```
In[149]:= Extract[exampleExpr, {1, 2}, Hold]
```

```
Out[149]= Hold[7 (3 + x)]
```

Note that we can use **{1,0}** to obtain the head of the expression, that is, the operator. This is equivalent to the use of Part ([[...]]) described above for the same purpose.

```
In[150]:= Extract[exampleExpr, {1, 0}]
```

```
Out[150]= Plus
```

```
In[151]:= exampleExpr[[1, 0]]
```

```
Out[151]= Plus
```

Combining Subtrees

As part of Huffman coding in Section 2, we wrote a function, **joinHTrees**, for joining two existing trees at a new root. We recreate that function here, with the weights removed.

```

In[152]:= joinTrees[newR_, A_?binaryTreeQ,
  B_?binaryTreeQ, opts___] := Module[
  {newRI, newT, newVerts, Aroot, Broot, newEdges, v, e, p, w},
  newRI = ToString[newRI];
  newVerts = Join[{newRI}, VertexList[A], VertexList[B]];
  Aroot = findRoot[A];
  Broot = findRoot[B];
  newEdges = Join[EdgeList[A], EdgeList[B],
    {DirectedEdge[newRI, Aroot], DirectedEdge[newRI, Broot]}];
  newT = Graph[newVerts, newEdges, opts];
  Do[PropertyValue[{newT, v}, "order"] =
    PropertyValue[{A, v}, "order"]
    , {v, VertexList[A]}];
  Do[PropertyValue[{newT, v}, "order"] =
    PropertyValue[{B, v}, "order"]
    , {v, VertexList[B]}];
  PropertyValue[{newT, Aroot}, "order"] = 1;
  PropertyValue[{newT, Broot}, "order"] = 2;
  PropertyValue[{newT, newRI}, "order"] = 0;
  Do[PropertyValue[{newT, v}, VertexLabels] =
    PropertyValue[{A, v}, VertexLabels]
    , {v, VertexList[A]}];
  Do[PropertyValue[{newT, v}, VertexLabels] =
    PropertyValue[{B, v}, VertexLabels]
    , {v, VertexList[B]}];
  PropertyValue[{newT, newRI}, VertexLabels] = newR;
  drawBinaryTree[newT]
]

```

The Main Function

With **joinTrees** and **newExpressionTree** from above, we are ready to write the function for turning infix expression into binary trees.

The function accepts a single argument, **expr**, the expression. We will give the function the Hold-First attribute, so that the expression will be held without our having to explicitly type Hold when executing the function. This holds the argument only until it is first evaluated, so we must immediately place a more permanent Hold on it, provided it was not already explicitly held, and assign it to **e**.

We first test the type of **e**. If it is an integer or a symbol, then we use **newExpressionTree** to create a new binary tree with **e** as the only vertex.

Otherwise, we're in the recursive case. We use Extract to determine the left and right operands and the operator. After recursive calls to the function to create the trees for the two operands, the subtrees are joined at the operator into the result tree.

Here is the function.

```

In[153]:= SetAttributes[expressionTree, HoldFirst];
expressionTree[expr_, opts___] :=
Module[{e, lhs, rhs, operator, lhsTree, rhsTree, result},
  If[expr[[0]] === Hold,
    e = expr,
    e = Hold[expr]
  ];
  If[e[[1, 0]] === Integer || e[[1, 0]] === Symbol,
    result = newExpressionTree[ReleaseHold[e], opts],
    lhs = Extract[e, {1, 1}, Hold];
    operator = Extract[e, {1, 0}];
    rhs = Extract[e, {1, 2}, Hold];
    lhsTree = expressionTree[lhs];
    rhsTree = expressionTree[rhs];
    result = joinTrees[operator, lhsTree, rhsTree, opts]
  ];
  drawBinaryTree[result]
]

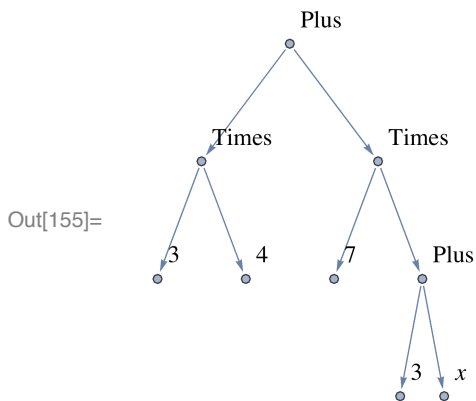
```

We test the function on the example expression.

```

In[155]:= exampleTree = expressionTree[3 * 4 + 7 * (3 + x), ImagePadding -> 10]

```



Prefix and Postfix Notation

Suppose we are given a binary tree representation of an arithmetic expression. We can express these trees in postfix, infix, or prefix form by applying the respective traversal algorithm we designed above. It is left to the reader to make the necessary functions to output infix, prefix, and postfix expressions. Be careful while doing so to keep in mind that the integers, variables, and operators are stored as the VertexLabels property of the vertices of the tree, not as the vertices themselves.

As a final example in this section, we demonstrate how to evaluate a given postfix expression. We will represent the postfix expression as a list of symbols, each of which is either a number or one of the arithmetic operations' symbols as a string.

Since we're considering postfix expressions, we read the list of symbols from left to right. Each time we encounter an operation, that operation is applied to the previous two numbers and we update the list by replacing the two numbers and the operation symbol by the result of the operation.

```
In[156]:= evalPostfix[expr_List] := Module[{i, L},
  L = expr;
  While[Length[L] > 1,
    i = 1;
    While[FreeQ[{"+", "-", "*", "/", "^"}, L[[i]]], i++];
    Switch[L[[i]],
      "+", L[[i]] = L[[i - 2]] + L[[i - 1]],
      "-", L[[i]] = L[[i - 2]] - L[[i - 1]],
      "*", L[[i]] = L[[i - 2]] * L[[i - 1]],
      "/", L[[i]] = L[[i - 2]] / L[[i - 1]],
      "^", L[[i]] = L[[i - 2]] ^ L[[i - 1]];
    L = Drop[L, {i - 2, i - 1}]
  ];
  L[[1]]
]
```

Note that the Drop function removes a range of elements of the list.

```
In[157]:= postfixExample = {7, 2, 3, "*", "-", 4, "^", 9, 3, "/", "+"}
```

```
Out[157]= {7, 2, 3, *, -, 4, ^, 9, 3, /, +}
```

```
In[158]:= evalPostfix[postfixExample]
```

```
Out[158]= 4
```

The reader is left to explore evaluation in the prefix case, which requires only a simple modification.

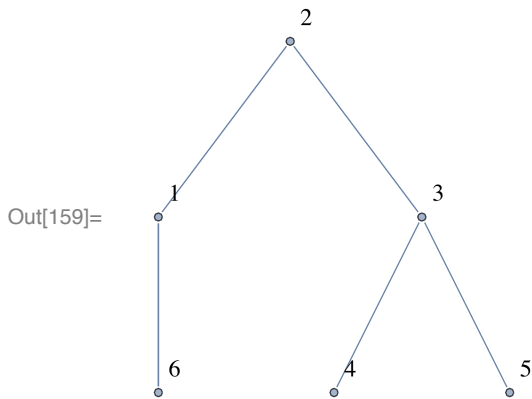
11.4 Spanning Trees

This section explains how to use *Mathematica* to construct spanning trees for graphs and how to use spanning trees to solve many different types of problems. Spanning trees have a myriad of applications, including coloring graphs, placing n queens on a $n \times n$ chessboard so that no two of the queens attack each other, and finding a subset of a set of numbers with a specified sum. All of these problems, which are described in detail in the text, will be explored computationally in this section. First we will show how to use *Mathematica* to form spanning trees using two algorithms: depth-first search and breadth-first search. Then we will show how to use *Mathematica* to solve the problems just mentioned.

Mathematica includes two functions, DepthFirstScan and BreadthFirstScan that will perform a search on a graph. Both functions accept a Graph and a vertex of the graph as the first two arguments. With these two arguments, the functions will return a list of vertices $\{v_1, v_2, \dots, v_n\}$. Suppose that the VertexList function returns $\{u_1, u_2, \dots, u_n\}$. Then the output of DepthFirstScan or BreadthFirstScan indicates that the tree whose edges are $v_i \leftrightarrow u_i$ is a spanning tree of the graph, with root the vertex such that $u_i = v_i$.

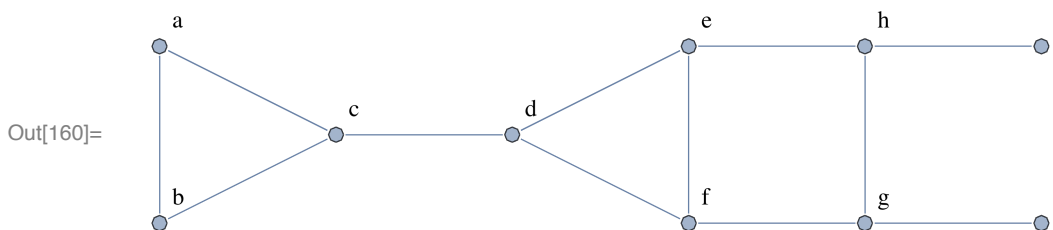
The function `TreeGraph` can be used to form a tree from the output of `DepthFirstScan` or `BreadthFirstScan`. `TreeGraph` can accept as input two lists of vertices, where the first is the list of all vertices to appear in the tree and the second is the list each of whose elements is the predecessor of the corresponding element of the first list. A simple example is shown below of a tree with root 2, vertices 1 and 3 are the children of 2, and 4 and 5 are the children of 3, and vertex 6 is the child of 1.

```
In[159]:= TreeGraph[{1, 2, 3, 4, 5, 6}, {2, 2, 2, 3, 3, 1},
  VertexLabels -> "Name", ImagePadding -> 10]
```



We will illustrate the use of `DepthFirstScan` or `BreadthFirstScan` with the graph from Exercise 13 of Section 11.4, which we recreate.

```
In[160]:= exercisel3 =
  Graph[{"a", "b", "c", "d", "e", "f", "g", "h", "i", "j"},
    {"a" -> "b", "a" -> "c", "b" -> "c", "c" -> "d", "d" -> "e",
      "d" -> "f", "e" -> "f", "e" -> "h", "f" -> "g", "g" -> "h",
      "g" -> "j", "h" -> "i"}, DirectedEdges -> False,
    VertexCoordinates -> {{0, 1}, {0, 0}, {1, 0.5}, {2, 0.5},
      {3, 1}, {3, 0}, {4, 0}, {4, 1}, {5, 1}, {5, 0}},
    VertexLabels -> "Name", ImagePadding -> 5]
```

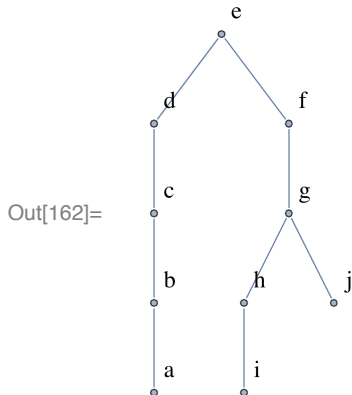


Using `DepthFirstScan` and passing the output to `TreeGraph` allows us to draw the spanning tree obtained by a depth-first search.

```
In[161]:= exercisel3DepthFirst = DepthFirstScan[exercisel3, "a"]
```

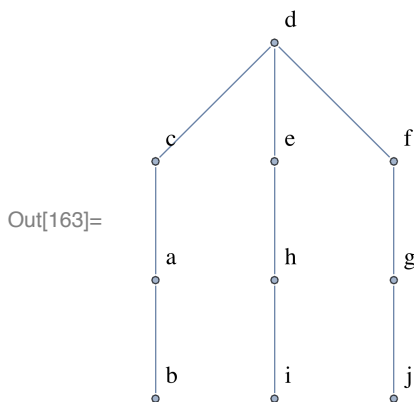
```
Out[161]= {a, a, b, c, d, e, f, g, h, g}
```

```
In[162]:= TreeGraph[VertexList[exercise13], exercise13DepthFirst,
  VertexLabels -> "Name", ImagePadding -> 10]
```



Observe that BreadthFirstScan produces a different tree.

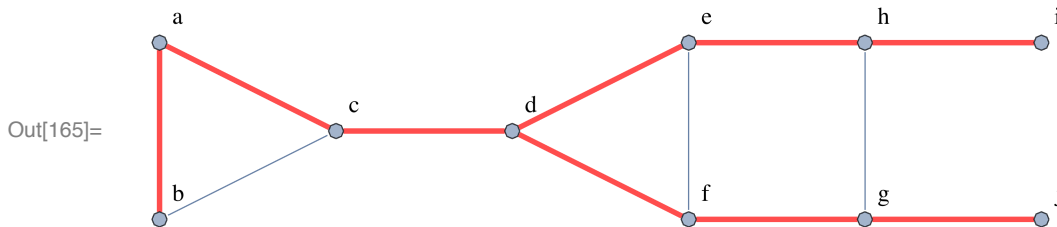
```
In[163]:= TreeGraph[VertexList[exercise13],
  BreadthFirstScan[exercise13, "a"],
  VertexLabels -> "Name", ImagePadding -> 10]
```



Note that in both cases, the initial vertex, *a*, is not drawn at the root of the tree. As we described previously, *Mathematica* draws trees to minimize depth, irrespective of the logical root.

DepthFirstScan and BreadthFirstScan both accept an optional third argument, which is given as a list of rules. The rules in the list identify an “event” with a pure Function (&) to be executed when the event occurs. We will illustrate this by highlighting the BreadthFirstScan tree within the graph. We make a copy of the graph first, since our execution of BreadthFirstScan will modify the graph.

```
In[164]:= exercisel3Copy = exercisel3;  
BreadthFirstScan[exercisel3Copy, "a", {"FrontierEdge" →  
  ((PropertyValue[{exercisel3Copy, #}, EdgeStyle] =  
    Directive[Thick, Red]) &)}]; exercisel3Copy
```



We now analyze the function executed above. `BreadthFirstScan` is called on the graph `exercisel3Copy` and told to start with vertex `a`. The third argument is the list containing a single Rule (`→`) identifying the event “FrontierEdge” with a pure Function (`&`). A frontier edge refers to an edge between a vertex that is currently being visited and a vertex that is newly discovered as a result of visiting the current vertex. The pure Function (`&`) associated to the event “FrontierEdge” is called with a single argument, the edge in question. In this example, we use the function to highlight the edges which are traversed by the breadth-first exploration.

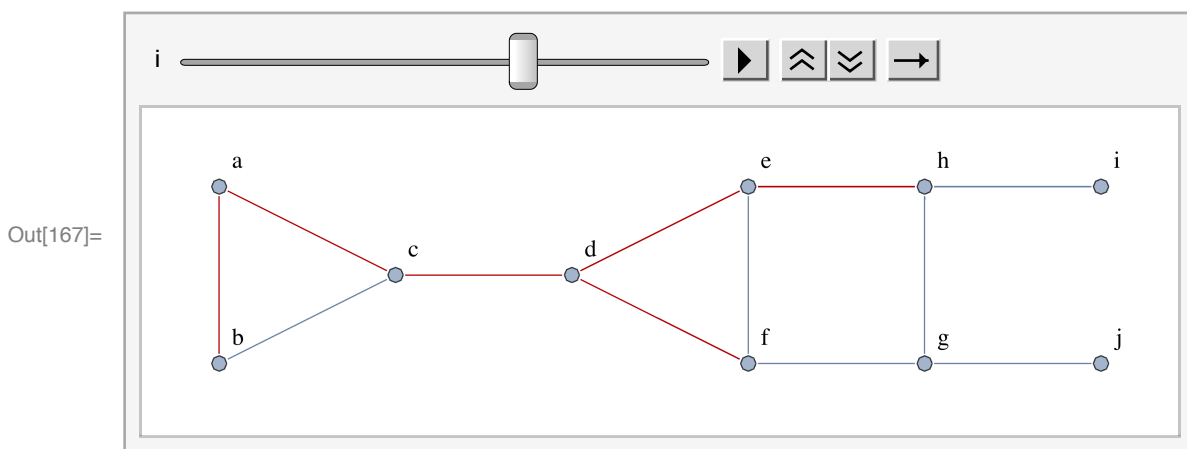
Alternately, we could use `Sow` and `Reap` to obtain a list of the edges in the order they are followed. Note that `Sow` accepts a single argument so it is not necessary to explicitly call it on a `Slot` (`#`).

```
In[166]:= exercisel3breadthList = Reap[BreadthFirstScan[  
  exercisel3, "a", {"FrontierEdge" → Sow}]] [[2, 1]]
```

Out[166]= {a ↔ b, a ↔ c, c ↔ d, d ↔ e, d ↔ f, e ↔ h, f ↔ g, h ↔ i, g ↔ j}

We can now use this list to create an animation.

```
In[167]:= Animate[  
  HighlightGraph[exercisel3, exercisel3breadthList[[1 ;; i]]],  
  {i, 0, Length[exercisel3breadthList], 1},  
  AnimationRunning → False, AnimationRepetitions → 1]
```



We leave it to the reader to create a function that generates such an animation automatically.

A second useful event that can be used with `DepthFirstScan` and `BreadthFirstScan` is “DiscoverVertex”. This event occurs when a vertex is discovered for the first time. The associated function is called with up to three arguments: the vertex being discovered, the vertex currently being visited, and the distance the visited vertex is from the starting vertex.

The interested reader is encouraged to consult the help pages for `DepthFirstScan` or `BreadthFirstScan` for information on other events that can be used.

Despite *Mathematica*'s existing functions, we will develop our own using depth-first and breadth-first search functions as a way to illustrate these important algorithms.

Depth-First Search

We begin by implementing depth-first search. As the name of the algorithm suggests, vertices are visited in order of increasing depth of the spanning tree. Our implementation is based on Algorithm 1 of Section 11.4 of the textbook.

Recall the terminology defined in the textbook. We say that we are “exploring a vertex” v from the time the vertex is first added to the spanning tree until we have backtracked back to v for the last time. Note that at any step in the process, we are generally exploring multiple vertices. In particular, the root of the spanning tree starts being explored at the very beginning of the process and continues being explored until it terminates.

The function, which we call **depthSearch**, will take two arguments: an undirected graph and a vertex in that graph. The function operates as follows:

1. First, we check that the graph is connected using *Mathematica*'s `ConnectedGraphQ` function. If not, there can be no spanning tree and the function returns `$Failed`.
2. Next, we initialize the following variables.
 - a. **toVisit** will be the set of vertices of the graph that have not yet been visited. It is initialized to the set of vertices of the graph minus the initial vertex.
 - b. **exploring** will be the list of vertices that are currently being explored. As vertices are visited, they are added to the end of the **exploring** list. When a vertex has been fully explored, i.e., when it has no neighbors not already in the tree, then we remove it from the **exploring** list. **exploring** is initialized to the vertex that is given as the second argument.
 - c. **T** will be the spanning tree that is constructed. It is initialized to the graph consisting of all the vertices of the graph, but with no edges. Provided that the graph is connected, we know that all the vertices will appear in **T** and this saves us from adding them one at a time.
3. Following initialization, we begin a `While` loop which terminates when the **exploring** list is empty. The variable **v** is set to the last element of the **exploring** list. We then compute the intersection, **N**, of the neighbors of **v** and the **toVisit** set of vertices not already contained in the tree. Either,
 - a. **N** is non-empty, in which case, one of its elements is chosen as **w**, the next vertex to visit. The edge $v \rightarrow w$ is added to the tree **T**. Also, **w** is removed from the **toVisit** set and added to the end of the **exploring** list. In the next iteration of the `While` loop, this new vertex will be set to be **v**.
 - b. **N** is empty, in which case the vertex **v** has been explored completely and so it can be removed from the **exploring** list. The next iteration of the `While` loop will set **v** to be the vertex one step back in the **exploring** list. This is the “backtracking” step.

Here, now, is the function.

```

In[168]:= depthSearch[G_?UndirectedGraphQ, startV_, opts___] :=
Module[{toVisit, exploring, T, v, N, w},
  If[! ConnectedGraphQ[G], Return[$Failed]];
  toVisit = Complement[VertexList[G], {startV}];
  exploring = {startV};
  T = Graph[VertexList[G], {}, opts];
  While[Length[exploring] > 0,
    v = exploring[[-1]];
    N = Intersection[AdjacencyList[G, v], toVisit];
    If[Length[N] > 0,
      w = N[[1]];
      T = EdgeAdd[T, v ↔ w];
      toVisit = Complement[toVisit, {w}];
      AppendTo[exploring, w],
      (*else*)
      exploring = Delete[exploring, -1]
    ]
  ];
  T
]

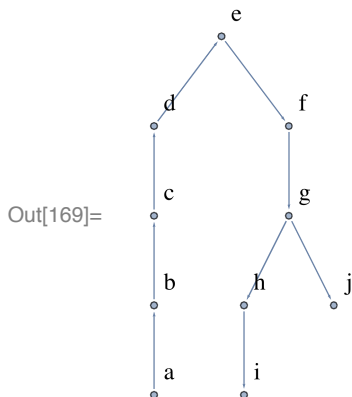
```

Let's test this with our Exercise 13 example from above.

```

In[169]:= depthExercisel3 = depthSearch[exercisel3,
  "a", VertexLabels → "Name", ImagePadding → 10]

```



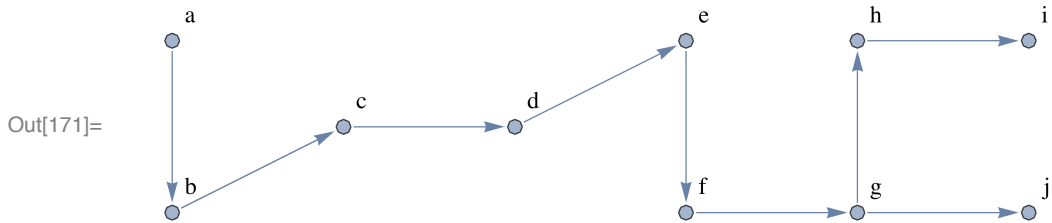
We can reposition the vertices to match the original with PropertyValue and VertexCoordinates.

```

In[170]:= Do[PropertyValue[{depthExercisel3, v}, VertexCoordinates] =
  PropertyValue[{exercisel3, v}, VertexCoordinates]
, {v, VertexList[exercisel3]}]

```

```
In[171]:= depthExercise13
```



Breadth-First Search

We now turn to an implementation of a breadth-first search. Recall that the breadth-first algorithm works by examining all vertices at the current depth of the spanning tree before moving on to the next level of the graph. Our implementation will follow Algorithm 2 of Section 11.4 of the text.

The function, to be called **breadthSearch**, again takes two arguments: an undirected graph and a vertex to act as the starting point. It proceeds as follows.

1. First, we check that the graph is connected using *Mathematica's* ConnectedGraphQ function.
2. Next, we initialize the following variables.
 - a. **toVisit**, as before, will be the set of vertices of the graph not yet visited. It is initialized to the set of vertices of the graph with the initial vertex excluded.
 - b. **toProcess** will be the list of vertices that have been determined to be incident to a vertex in the tree but which have not yet been processed. **toProcess** is initialized to the vertex that is given as the second argument to the procedure.
 - c. **T** will be the spanning tree that is constructed. Once again, it is initialized to the tree consisting of all the vertices of the given graph, but with no edges.
3. Following initialization, we begin a while loop that terminates when the **toProcess** list is empty. The variable **v** is set to the *first* element of the **toProcess** list. We then compute the intersection, **N**, of the neighbors of **v** and the **toVisit** set. For each element **w** of **N**, an edge **v**→**w** is added to **T** and **w** is added to the end of the **toProcess** list and removed from the **toVisit** set. Then **v** is removed from **toProcess**.

Observe that, since neighbors are added to the end of the **toProcess** list and are processed from the beginning of the list, we are assured that all vertices on a given level will be processed before any vertex at a lower level.

Here is the implementation.

```

In[172]:= breadthSearch[G_?UndirectedGraphQ, startV_, opts___] :=
Module[{toVisit, toProcess, T, v, N, w},
  If[! ConnectedGraphQ[G], Return[$Failed]];
  toVisit = Complement[VertexList[G], {startV}];
  toProcess = {startV};
  T = Graph[VertexList[G], {}, opts];
  While[Length[toProcess] > 0,
    v = toProcess[[1]];
    N = Intersection[AdjacencyList[G, v], toVisit];
    Do[
      T = EdgeAdd[T, v → w];
      AppendTo[toProcess, w];
      toVisit = Complement[toVisit, {w}];
      , {w, N}];
    toProcess = Delete[toProcess, 1];
  ];
  T
]

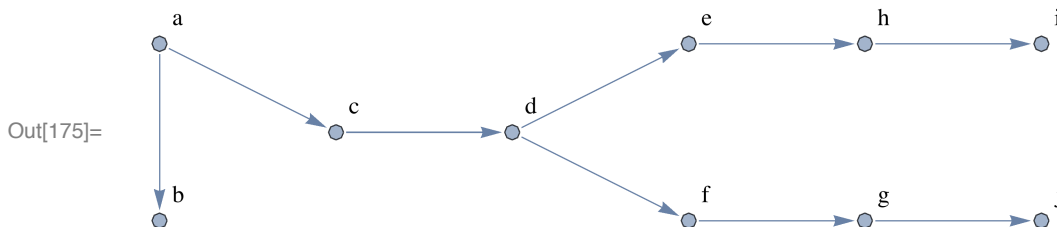
```

Once again, we illustrate using Exercise 13.

```

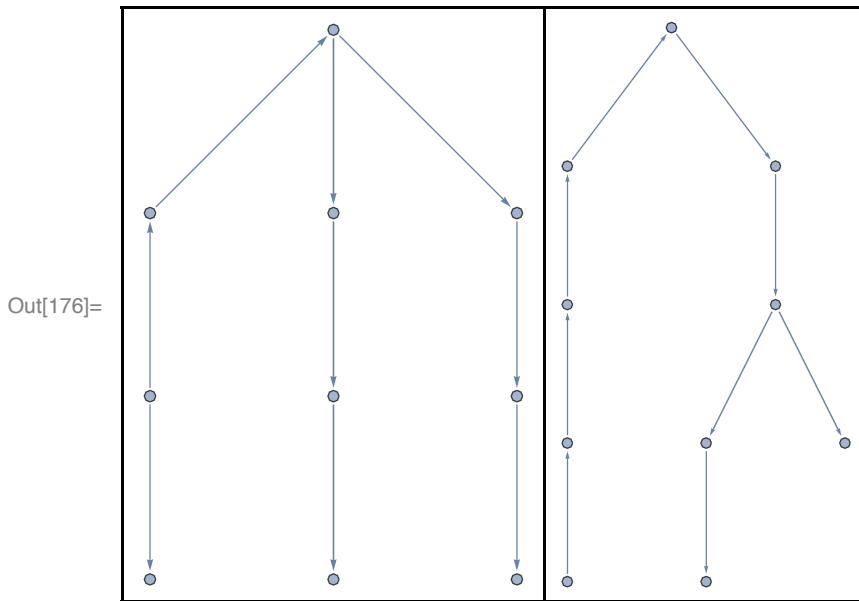
In[173]:= breadthExercise13 = breadthSearch[exercise13,
  "a", VertexLabels → "Name", ImagePadding → 5];
Do[PropertyValue[{breadthExercise13, v}, VertexCoordinates] =
  PropertyValue[{exercise13, v}, VertexCoordinates]
, {v, VertexList[exercise13]}];
breadthExercise13

```



Before moving on to backtracking, let's take a moment to compare the trees produced by the two algorithms.

```
In[176]:= Grid[{ {breadthSearch[exercise13, "a"],
                  depthSearch[exercise13, "a"]} }, Frame -> All]
```



Notice that the two spanning trees are quite different, even though *Mathematica* roots them to be as balanced as possible. In particular, the depth-first search has a deep and thin structure, whereas the breadth-first search is shorter and wider appearing.

Graph Coloring via Backtracking

Backtracking is a method that can be used to find solutions to problems that might be impractical to solve using exhaustive search techniques. Backtracking is based on the systematic search for a solution to a problem using a decision tree. (See the text for a complete discussion.) Here we show how to use backtracking to solve several different problems, including coloring a graph, the n -queens problem, and the subset sum problem.

The first problem we will attack via a backtracking procedure is the problem of coloring a graph using n colors, where n is a positive integer. Given a graph, we will attempt to color it using n colors using the method described in Example 6 of Section 11.4.

1. Fix an order on the vertices of the graph, say v_1, v_2, \dots, v_m and fix an ordering of the colors as color 1, color 2, ..., color n . We will use the ordering of the vertices that *Mathematica* automatically imposes. For the colors, we will require an ordered list of colors as one of the arguments to the function.
2. We store the current state of the coloring in a list we will call **coloring**. The i th entry in this list will correspond to the color of the i th vertex. For example, **coloring** = {1, 2, 1}, corresponds to vertex v_1 assigned color 1, vertex v_2 assigned color 2, and vertex v_3 assigned color 1. This **coloring** list is similar to the **exploring** list from **depthSearch**. In both cases, you can think of the list as storing the path from the root of the tree to the current vertex. In this case, the level, which corresponds to the position in the list, carries additional information. Specifically, level k in the decision tree (i.e., position k in the **coloring** list) corresponds to deciding the color of vertex v_k .

3. We initialize **coloring** to {1} and set a counter variable **i** to 2. The variable **i** will indicate the vertex that requires a decision.
4. Set **N** equal to the neighbors of the *i*th vertex of the graph, and then construct a set, **used**, consisting of the indices of the colors assigned to the neighbors. The *i*th vertex will be assigned the color with the smallest index not in **used**, assuming there are any remaining colors.
5. If there are no possible colors for the *i*th vertex, then we must backtrack. We decrease *i* by one. To ensure that we do not repeat a choice already made when we revisit a vertex, we make the following modification to how colors are chosen. If the *i*th position of **coloring** has already been set, then we know we're in the process of backtracking. We insist that the new choice for the color of vertex *i* is the smallest possible color greater than the current color.
6. The function terminates in one of two cases. If **i** is set to a value greater than the number of vertices, then we know that **coloring** contains a valid assignment for all vertices. On the other hand, if **i** is ever set to 1, then we know that we have backtracked all the way to the root. Since the color of the first vertex doesn't affect the validity of the coloring, this indicates that we have exhausted all possible colorings and that the graph cannot be colored with *n* colors.

Our function will be called **backColor**. It will accept two arguments: the graph to be colored and a list of colors. If it is successful, it will display the graph with the vertices colored. If it determines that there is no *n*-coloring of the graph, it will return **Failed**.

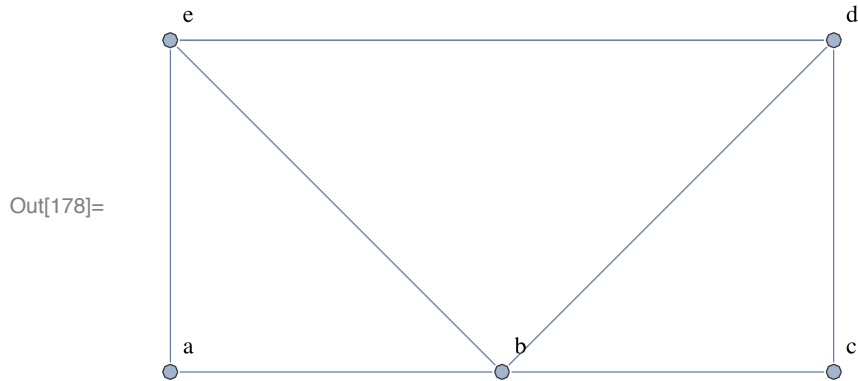
```

In[177]:= backColor[G_Graph, C : {__RGBColor}] :=
Module[{verts, numverts, allcolorsL,
  k, coloring, i, N, j, used, available, g},
  verts = VertexList[G];
  numverts = Length[verts];
  allcolorsL = Range[Length[C]];
  coloring = {1};
  i = 2;
  While[i > 1 && i ≤ numverts,
    N = VertexList[NeighborhoodGraph[G, verts[[i]]]];
    used = {};
    For[j = 1, j ≤ i - 1, j++,
      If[MemberQ[N, verts[[j]]], AppendTo[used, coloring[[j]]]];
    ];
    If[Length[coloring] ≥ i,
      used = Union[used, Range[coloring[[i]]]];
      available = Complement[allcolorsL, used];
      If[Length[available] > 0,
        coloring = Append[coloring[[1 ;; i - 1]], First[available]];
        i++,
        (*else*)
        If[Length[coloring] ≥ i, coloring = coloring[[1 ;; i - 1]];
        i--
      ];
    ];
  ];
  If[i > numverts,
    g = G;
    For[k = 1, k ≤ numverts, k++,
      PropertyValue[{g, verts[[k]]}, VertexStyle] =
        C[[coloring[[k]]]];
    ];
    Return[g],
    (*else*)
    Return[$Failed]
  ]
]

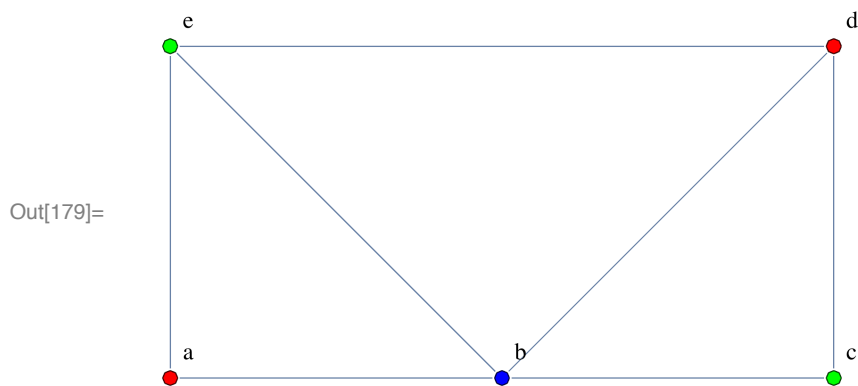
```

We test our function on the example given in Figure 11 of Section 11.4 of the text.

```
In[178]:= figure11Graph = Graph[{"a", "b", "c", "d", "e"},
  {"a" → "b", "a" → "e", "b" → "c", "b" → "d", "b" → "e",
   "c" → "d", "d" → "e"}, DirectedEdges → False,
  VertexCoordinates → {{0, 0}, {1, 0}, {2, 0}, {2, 1}, {0, 1}},
  VertexLabels → "Name", ImagePadding → 10]
```



```
In[179]:= backColor[figure11Graph, {Red, Blue, Green}]
```



On the other hand, the complete graph on 5 vertices cannot be 3-colored or 4-colored.

```
In[180]:= K5 = CompleteGraph[5];
```

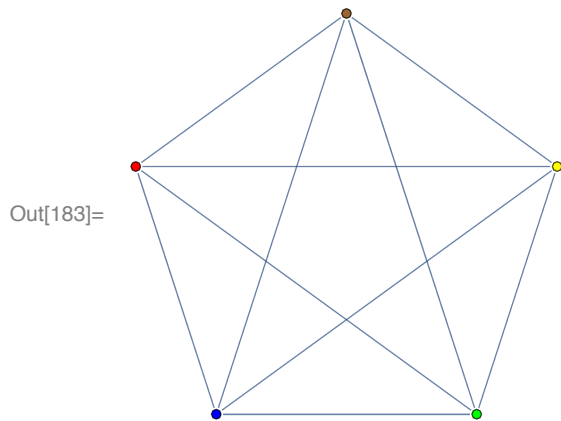
```
In[181]:= backColor[K5, {Red, Blue, Green}]
```

Out[181]= \$Failed

```
In[182]:= backColor[K5, {Red, Blue, Green, Yellow}]
```

Out[182]= \$Failed

```
In[183]:= backColor[K5, {Red, Blue, Green, Yellow, Brown}]
```



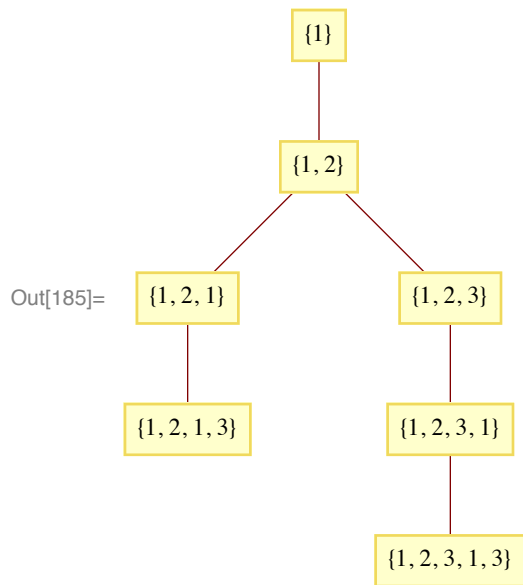
Before moving on to the n -queens problem, we illustrate how we can modify our backtracking function to record and display the decision tree. Instead of displaying the graph, our modified algorithm will produce the decision tree. We do this by building a list of the decisions. Each time a color is assigned, we create a new vertex by adding the current state of the coloring list (converted to a string) and connecting it to the previous state. Note that we use TreePlot here, rather than a Graph object, so as to be able to draw the root at the top of the image.

```

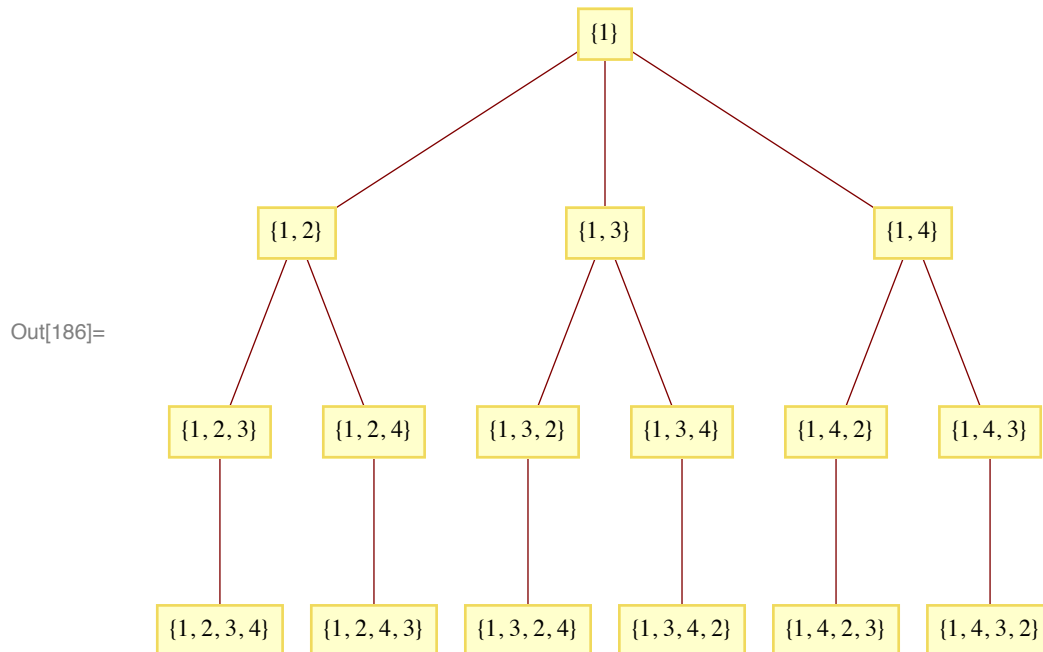
In[184]:= backColorDT[G_Graph, C : {__RGBColor}] :=
Module[{verts, numverts, allcolorsL, k, coloring,
  i, N, j, used, available, DTList, parentV, newV},
verts = VertexList[G];
numverts = Length[verts];
allcolorsL = Range[Length[C]];
coloring = {1};
newV = ToString[coloring];
DTList = {};
i = 2;
While[i > 1 && i ≤ numverts,
  N = VertexList[NeighborhoodGraph[G, verts[[i]]]];
  used = {};
  For[j = 1, j ≤ i - 1, j++,
    If[MemberQ[N, verts[[j]]], AppendTo[used, coloring[[j]]]];
  ];
  If[Length[coloring] ≥ i,
    used = Union[used, Range[coloring[[i]]]];
    available = Complement[allcolorsL, used];
    If[Length[available] > 0,
      parentV = ToString[coloring[[1 ;; i - 1]]];
      coloring =
        Append[coloring[[1 ;; i - 1]], First[available]];
      newV = ToString[coloring];
      AppendTo[DTList, parentV → newV];
      i++,
      (*else*)
      If[Length[coloring] ≥ i, coloring = coloring[[1 ;; i - 1]];
      i--
    ]
  ];
TreePlot[DTList, Top, "{1}", VertexLabeling → True]
]

```

```
In[185]:= backColorDT[figure11Graph, {Red, Blue, Green}]
```



```
In[186]:= backColorDT[K5, {Red, Blue, Green, Yellow}]
```



n-Queens Problem via Backtracking

Another problem with an elegant backtracking solution is the problem of placing n queens on an $n \times n$ chessboard so that no queen can attack another. This means that no two queens can be placed on the same horizontal, vertical, or diagonal line. We will solve this problem with a backtracking algorithm. The solution we present here is based on the solution given in Example 7 in Section 11.4. We place queens in a greedy fashion on the chessboard until either all the queens are placed or there is no available position for a queen to be placed without coming under attack from a queen already on the board.

Following the textbook, the i th step in the backtracking algorithm will be to place a queen in the i th column (or file, in chess terms). Like the **coloring** list and **exploring** list, the algorithm will build a **queens** list. In this case, **queens**[[i]] = j will indicate that a queen is placed in the j th row (rank) in the i th column (file). We will build a helper function, **validQueens**, that, given the dimension of the board and the current **queens** list, will determine the possible locations for a queen in the next column.

To implement **validQueens**, we will need a representation of the status of the board; specifically, for each square on the board, whether it is safe or under attack. It is natural to represent the board as a matrix with an entry 1 indicating that the corresponding square is safe and 0 that it is under attack. We create a square matrix with all entries initialized to 1 by issuing the ConstantArray function with two arguments: the common value 1 and the dimension of the matrix in a list. For example, the following initializes the matrix representing the board of dimension 5 on which no queens have been placed.

```
In[187]:= ConstantArray[1, {5, 5}] // MatrixForm
```

```
Out[187]//MatrixForm=
```

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

We now build a **boardStatus** function that, given the current list of queen locations and the dimension of the board will return a matrix representing the board with that configuration. The matrix will contain 1 in positions not under attack, 0 in positions under attack, and will represent the location of queens by the symbol Q.

```

In[188]:= boardStatus[queenList_List, dim_Integer] :=
Module[{board, i, dif, qRank, qFile, vQueens},
  board = ConstantArray[1, {dim, dim}];
  For[qFile = 1, qFile ≤ Length[queenList], qFile++,
    qRank = queenList[[qFile]];
    For[i = 1, i ≤ dim, i++,
      board[[qRank, i]] = 0;
      board[[i, qFile]] = 0;
      dif = i - qFile;
      If[qRank + dif ≤ dim && qRank + dif ≥ 1,
        board[[qRank + dif, i]] = 0];
      If[qRank - dif ≤ dim && qRank - dif ≥ 1,
        board[[qRank - dif, i]] = 0]
    ]
  ];
  For[qFile = 1, qFile ≤ Length[queenList], qFile++,
    board[[queenList[[qFile]], qFile]] = "Q"
  ];
  board
]

```

For example, on a 10×10 board, with the first queen in the second row and the second queen in the seventh row, the board looks as follows.

```

In[189]:= boardStatus[{2, 7}, 10] // MatrixForm

```

Out[189]//MatrixForm=

$$\begin{pmatrix}
 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
 Q & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\
 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\
 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
 0 & Q & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\
 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1
 \end{pmatrix}$$

The **validQueens** function will take the same arguments, the list of queen locations and dimension of the board, pass them to **boardStatus**, and use the resulting matrix to determine available positions in the next column. We could omit the **boardStatus** function and instead create **validQueens** independently, but, as you see above, the **boardStatus** function provides a useful visualization.

```

In[190]:= validQueens[queenList_List, dim_Integer] :=
Module[{board, file, i, freeSet},
  board = boardStatus[queenList, dim];
  file = Length[queenList] + 1;
  freeSet = {};
  For[i = 1, i ≤ dim, i++,
    If[board[[i, file]] == 1, AppendTo[freeSet, i]]
  ];
  freeSet
]

```

With this preliminary work out of the way, we are ready to write the main program, **nQueens**. It will work in much the same way as our previous examples.

1. We keep a **queens** list, initialized to the empty list, that records the locations of queens.
2. We initialize a counter **file** to 1. This indicates the column in which we need to place a queen. Notice that, in the **backColor** algorithm, we initialized the counter to 2. The reason for the difference is that, in the coloring algorithm, the color of the first vertex was arbitrary and changing it from color 1 to a different color could not possibly affect the outcome. In this case, it may be the case that there is no solution with the first queen in file 1, rank 1, but there is a solution if the file 1 queen is in a different rank.
3. Apply the **validQueens** function with the **queens** list and the board dimension. Store the resulting set as **open**.
4. As with the **backColor** algorithm, we determine if the current assignment is a new assignment or a result of backtracking. If it is a backtracking step, we remove from **open** the positions equal to or smaller than the previous attempt.
5. We terminate when **file** either exceeds the board dimension, in which case we have found a solution, or when it is backtracked to 0, in which case we have exhausted all possibilities.

```

In[191]:= nQueens[boardDim_Integer] /; boardDim > 0 :=
Module[{queens = {}, file = 1, open, i},
While[file > 0 && file ≤ boardDim,
open = validQueens[queens[[1 ;; file - 1]], boardDim];
If[Length[queens] ≥ file,
open = Complement[open, Range[queens[[file]]]]];
If[open ≠ {},
queens = Join[queens[[1 ;; file - 1]], {open[[1]]}];
file++,
(*else backtrack*)
queens = queens[[1 ;; file - 1]];
file--
]
];
If[file > boardDim,
Return[MatrixForm[boardStatus[queens, boardDim]]],
Return[$Failed]
]
]

```

We can use this to find one solution to the 8-queens problem (8×8 is the size of the standard board).

```

In[192]:= nQueens[8]

```

```

Out[192]//MatrixForm=

```

$$\begin{pmatrix} Q & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & Q & 0 \\ 0 & 0 & 0 & 0 & Q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & Q \\ 0 & Q & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & Q & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & Q & 0 & 0 \\ 0 & 0 & Q & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Subset Sum Problem via Backtracking

Finally, we consider the subset sum problem. Given a set of integers S and a value M , we want to find a subset B of S whose sum is M . To use backtracking on this problem, we first impose an ordering on the set S . We successively select integers from S to include in B until the sum of the elements of B equals or exceeds M , and backtrack when the sum exceeds M .

Before we get to the main algorithm, it is worth reviewing two items of syntax. First, given a list of values, we can compute their sum with the `Plus` function by using `Apply (@@)` to replace the `List` head with the `Plus` head, which then results in the sum being computed.

```

In[193]:= listofvalues = {3, 7, 11, 15, -4}

```

```

Out[193]= {3, 7, 11, 15, -4}

```

```
In[194]:= Plus @@ listofvalues
```

```
Out[194]= 32
```

Second, given a list of values and a second list consisting of indices into the first list, we can obtain the sublist of values corresponding to the positions described by the second list by using the Part (`[[...]]`) operator with the list of indices.

```
In[195]:= listofstuff = {"a", "b", "c", "d", "e", "f", "g"}
```

```
Out[195]= {a, b, c, d, e, f, g}
```

```
In[196]:= listofindices = {1, 3, 4, 7}
```

```
Out[196]= {1, 3, 4, 7}
```

```
In[197]:= listofstuff[[listofindices]]
```

```
Out[197]= {a, c, d, g}
```

```
In[198]:= listofstuff[{{3, 4, 6}}]
```

```
Out[198]= {c, d, f}
```

As the general pattern of backtracking algorithms should be clear by this point, we omit a detailed description of the function.

```

In[199]:= subsetSum[S : {__Integer}, M_Integer] :=
Module[{bIndices, allIndices, i, availIndices, k, currentSum},
  bIndices = {};
  allIndices = Range[Length[S]];
  i = 1;
  currentSum = 0;
  While[i > 0 && currentSum ≠ M,
    availIndices = Complement[allIndices, bIndices];
    If[Length[bIndices] ≥ i, availIndices =
      Complement[availIndices, Range[bIndices[[i]]]];
    Do[If[currentSum + S[[k]] > M,
      availIndices = Complement[availIndices, {k}]
      , {k, availIndices}];
    If[availIndices ≠ {},
      bIndices =
        Append[bIndices[[1 ;; i - 1]], availIndices[[1]]];
      i++,
      (*else*)
      If[Length[bIndices] ≥ i, bIndices = bIndices[[1 ;; i - 1]];
      i--
    ];
    currentSum = Plus @@ S[[bIndices[[1 ;; i - 1]]]
  ];
  If[i == 0,
    Return[$Failed],
    Return[S[[bIndices]]]
  ]
]

```

```

In[200]:= subsetSum[{31, 27, 15, 11, 7, 5}, 39]

```

```

Out[200]= {27, 7, 5}

```

```

In[201]:= subsetSum[{31, 27, 15, 11, 7, 5}, 40]

```

```

Out[201]= $Failed

```

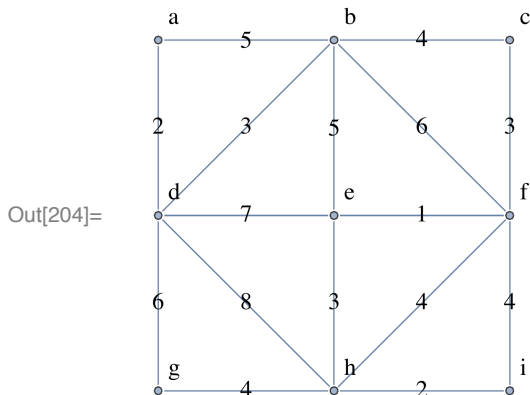
11.5 Minimum Spanning Trees

This section explains how to use *Mathematica* to find the minimum spanning tree of a weighted graph. Recall that a minimum spanning tree T of a weighted graph G is a spanning tree of G with the minimum weight of all spanning trees of G . The two best known algorithms for constructing minimum spanning trees are called Prim's algorithm and Kruskal's algorithm. In this section, we will implement these algorithms, as this is another case in which understanding the implementation can help you better understand the algorithms.

Before implementing the algorithms and finding minimum spanning trees, though, we will describe two built-in *Mathematica* commands, GraphDistance and GraphDistanceMatrix, that calculate the minimum distance between two vertices in a graph.

First, we construct a graph to use as an example. We will recreate Exercise 3 from Section 11.5. To define undirected, weighted edges, we will give EdgeWeight as an option to Graph, associating it with the list of the edge weights in the same order as the edges are given. Recall that to get the edge weights to appear as labels on the graph, we need to explicitly set the edge labels, which we do by constructing a loop setting the label of each edge to its weight.

```
In[202]:= exercise3 = Graph[{"a", "b", "c", "d", "e", "f", "g", "h", "i"},
  {"a" → "b", "a" → "d", "b" → "c", "b" → "d",
   "b" → "e", "b" → "f", "c" → "f", "d" → "e", "d" → "g",
   "d" → "h", "e" → "f", "e" → "h", "f" → "h", "f" → "i",
   "g" → "h", "h" → "i"}, DirectedEdges → False,
  EdgeWeight → {5, 2, 4, 3, 5, 6, 3, 7, 6, 8, 1, 3, 4, 4, 2},
  VertexCoordinates → {{0, 2}, {1, 2}, {2, 2},
    {0, 1}, {1, 1}, {2, 1}, {0, 0}, {1, 0}, {2, 0}},
  VertexLabels → "Name", ImagePadding → 10];
Do[PropertyValue[{exercise3, e}, EdgeLabels] =
  PropertyValue[{exercise3, e}, EdgeWeight],
  {e, EdgeList[exercise3]};
exercise3
```



Mathematica's GraphDistance function accepts two or three arguments. Given a graph and two vertices in the graph, it returns the distance between the two vertices. For example, in the graph above, we compute the distance from *c* to *e* as follows.

```
In[205]:= GraphDistance[exercise3, "c", "e"]
```

Out[205]= 4.

You can also apply GraphDistance to a graph and just one of its vertices. In this case, the output will be a list of the shortest distances from the given vertex to each of the vertices in the graph. For example, the following shows the distances from the central vertex *e* to each of the vertices in the graph. The 0 in the output corresponds to the distance from the vertex to itself.

```
In[206]:= GraphDistance[exercise3, "e"]
```

```
Out[206]= {9., 5., 4., 7., 0, 1., 7., 3., 5.}
```

The GraphDistanceMatrix function applied to a graph produces the matrix whose entries are the distances between corresponding vertices. The rows and columns are ordered according to the order of vertices in the graph, agreeing with the output of VertexList.

```
In[207]:= GraphDistanceMatrix[exercise3] // MatrixForm
```

```
Out[207]//MatrixForm=
```

$$\begin{pmatrix} 0. & 5. & 9. & 2. & 9. & 10. & 8. & 10. & 12. \\ 5. & 0. & 4. & 3. & 5. & 6. & 9. & 8. & 10. \\ 9. & 4. & 0. & 7. & 4. & 3. & 11. & 7. & 7. \\ 2. & 3. & 7. & 0. & 7. & 8. & 6. & 8. & 10. \\ 9. & 5. & 4. & 7. & 0. & 1. & 7. & 3. & 5. \\ 10. & 6. & 3. & 8. & 1. & 0. & 8. & 4. & 4. \\ 8. & 9. & 11. & 6. & 7. & 8. & 0. & 4. & 6. \\ 10. & 8. & 7. & 8. & 3. & 4. & 4. & 0. & 2. \\ 12. & 10. & 7. & 10. & 5. & 4. & 6. & 2. & 0. \end{pmatrix}$$

Using TableForm and the TableHeadings option, we can easily create a table of the distances with row and column headings making the vertices explicit. TableHeadings is associated with a list of two lists in order to label the rows and columns.

```
In[208]:= TableForm[GraphDistanceMatrix[exercise3], TableHeadings →  
{VertexList[exercise3], VertexList[exercise3]}]
```

```
Out[208]//TableForm=
```

	a	b	c	d	e	f	g	h	i
a	0.	5.	9.	2.	9.	10.	8.	10.	12.
b	5.	0.	4.	3.	5.	6.	9.	8.	10.
c	9.	4.	0.	7.	4.	3.	11.	7.	7.
d	2.	3.	7.	0.	7.	8.	6.	8.	10.
e	9.	5.	4.	7.	0.	1.	7.	3.	5.
f	10.	6.	3.	8.	1.	0.	8.	4.	4.
g	8.	9.	11.	6.	7.	8.	0.	4.	6.
h	10.	8.	7.	8.	3.	4.	4.	0.	2.
i	12.	10.	7.	10.	5.	4.	6.	2.	0.

Prim's Algorithm

We will now build functions to implement Prim's algorithm. We will also see how to create animations that illustrate the process of building the spanning tree.

Since both Prim's algorithm and Kruskal's depend on choosing an edge of smallest weight, it will be useful to be able to apply Sort to a list of edges. Recall that Sort accepts an optional second argument: a function that takes 2 arguments and returns True if the first argument is less than the second. Since this function needs to also depend on the graph, as well as the two edges, we will create a func-

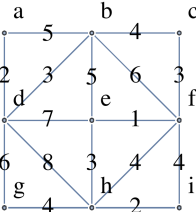
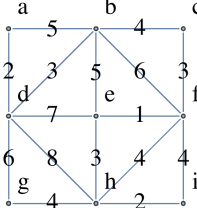
tion that takes a graph as the argument and produces a pure Function (&) that requires only the two edges as arguments.

```
In[209]:= edgeCompare[G_] := Function[{e1, e2},
      PropertyValue[{G, e1}, EdgeWeight] <
      PropertyValue[{G, e2}, EdgeWeight]
    ]
```

The list we provided to Function as its first argument specifies that the symbols **e1** and **e2** will be interpreted as parameters, in place of the generic Slot (#).

Applying **edgeCompare** to a graph will output a function. Assigning the result to a symbol means that symbol can then be used as a function. The expression below will cause **testComp** to act as a function on two argument, edges in **exercise3**, and return True if the weight of the first is less than or equal to the weight of the second.

```
In[210]:= testComp = edgeCompare[exercise3]
```

```
Out[210]= Function[{e1$, e2$},
      PropertyValue[{

, e1$}, EdgeWeight] <
      PropertyValue[{

, e2$}, EdgeWeight]]
```

```
In[211]:= testComp["a" ↔ "d", "a" ↔ "b"]
```

```
Out[211]= True
```

Prim's algorithm is given as Algorithm 1 in Section 11.5 of the textbook. It constructs a minimum spanning tree by successively selecting an edge of smallest weight that extends the tree without creating any loops.

To simplify our implementation of Prim's algorithm, we will create a function **minEdge**. Given the original graph and the list of vertices already included in the spanning tree, **minEdge** determines which edge of the graph should be added next.

minEdge first needs to determine the set of edges that are incident with a vertex currently in the tree. It will do this by applying NeighborhoodGraph to the graph and the list of vertices in the tree. The **minEdge** function then eliminates any edge with both ends already in the spanning tree or neither end in the spanning tree. (This is equivalent to the condition that the edge not introduce a simple circuit but

that the new graph will be connected.) Once it has determined the valid candidates, the **minEdge** function returns the edge with smallest weight. We also include the special case that the spanning tree has not yet been started, in which case we call the function with the empty list as the second argument.

```
In[212]:= minEdge[G_Graph, V_List] := Module[{possibleEdges, i},
  If[V == {},
    (* empty tree case *)
    possibleEdges = EdgeList[G],
    (* nonempty case *)
    possibleEdges = EdgeList[NeighborhoodGraph[G, V]];
  i = 1;
  While[i ≤ Length[possibleEdges],
    If[MemberQ[V, possibleEdges[[i]][[1]]] ==
      MemberQ[V, possibleEdges[[i]][[2]]],
      possibleEdges = Delete[possibleEdges, i],
      i++]
  ]
];
If[possibleEdges == {}, Return[Null]];
possibleEdges = Sort[possibleEdges, edgeCompare[G]];
possibleEdges[[1]]
]
```

With this function in place, Prim's algorithm is fairly straightforward to implement.

1. Begin building the spanning tree by finding the edge of minimum weight by calling the function **minEdge** on the graph and the empty list.
2. Continue building the spanning tree one edge at a time by adding the edge returned by **minEdge**. (Note that we must add the new vertex before the edge, since EdgeAdd expects both endpoints of the edge to be added to already be in the graph.)
3. After $n - 2$ repetitions of step 2, where n is the number of vertices in the graph, the spanning tree is complete.
4. For the sake of displaying the resulting tree, we conclude the function by copying the vertex positions from the original graph to the tree.

```

In[213]:= prim[G_Graph, opts___] := Module[{newEdge, T, n, i, v},
  newEdge = minEdge[G, {}];
  T = Graph[{newEdge}, opts];
  PropertyValue[{T, newEdge}, EdgeWeight] =
    PropertyValue[{G, newEdge}, EdgeWeight];
  PropertyValue[{T, newEdge}, EdgeLabels] =
    PropertyValue[{G, newEdge}, EdgeWeight];
  n = VertexCount[G];
  For[i = 1, i ≤ n - 2, i++,
    newEdge = minEdge[G, VertexList[T]];
    If[VertexQ[T, newEdge[[1]]],
      T = VertexAdd[T, newEdge[[2]]],
      T = VertexAdd[T, newEdge[[1]]]
    ];
    T = EdgeAdd[T, newEdge];
    PropertyValue[{T, newEdge}, EdgeWeight] =
      PropertyValue[{G, newEdge}, EdgeWeight];
    PropertyValue[{T, newEdge}, EdgeLabels] =
      PropertyValue[{G, newEdge}, EdgeWeight]
  ];
  Do[PropertyValue[{T, v}, VertexCoordinates] =
    PropertyValue[{G, v}, VertexCoordinates];
    PropertyValue[{T, v}, VertexLabels] =
      PropertyValue[{G, v}, VertexLabels]
    , {v, VertexList[T]}];
  T
]

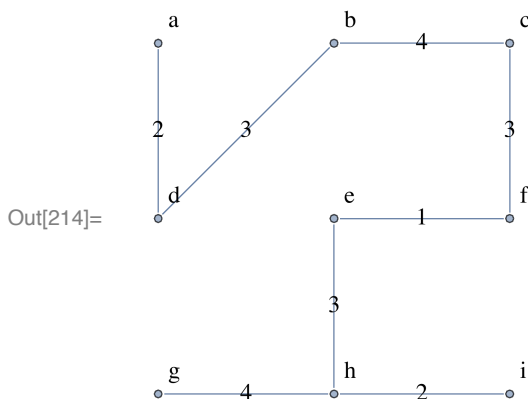
```

We can now use this algorithm to find the minimum spanning tree for Exercise 3.

```

In[214]:= primExercise3 = prim[exercise3, ImagePadding → 10]

```



Before moving on to Kruskal's algorithm, we'll create a function to produce an animation demonstrat-

ing Prim's algorithm in action. First, we'll modify the **prim** function to record the list of edges that form the tree and return this list of edges rather than the tree.

```
In[215]:= primEdges[G_Graph] := Module[{newEdge, T, edgeList, n, i, v},
  newEdge = minEdge[G, {}];
  T = Graph[{newEdge}];
  edgeList = {newEdge};
  n = VertexCount[G];
  For[i = 1, i ≤ n - 2, i++,
    newEdge = minEdge[G, VertexList[T]];
    If[VertexQ[T, newEdge[[1]]],
      T = VertexAdd[T, newEdge[[2]]],
      T = VertexAdd[T, newEdge[[1]]]
    ];
    T = EdgeAdd[T, newEdge];
    AppendTo[edgeList, newEdge]
  ];
  edgeList
]
```

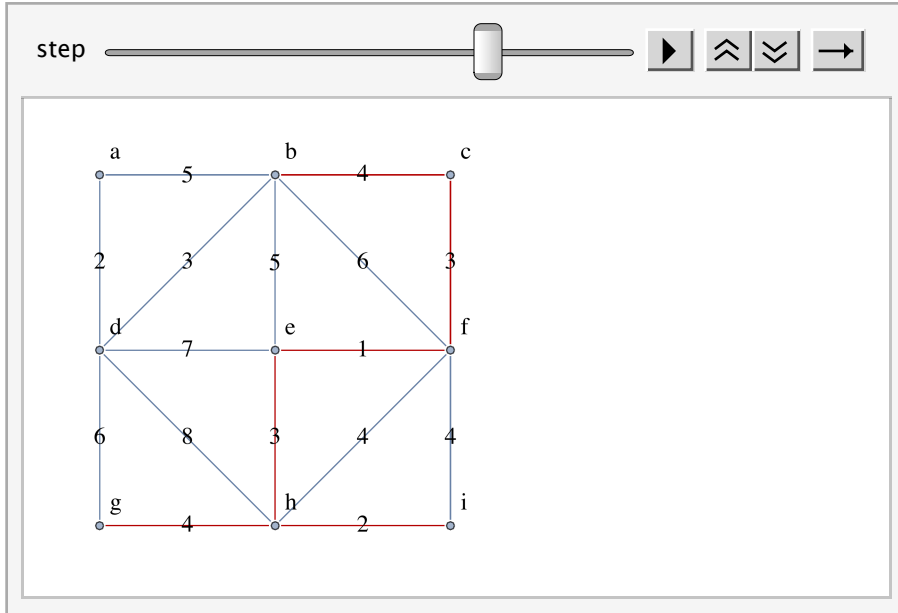
```
In[216]:= primEdges[exercise3]
```

```
Out[216]:= {e ↔ f, c ↔ f, e ↔ h, h ↔ i, g ↔ h, b ↔ c, b ↔ d, a ↔ d}
```

We now write the function that will produce the animation. Since this is nearly identical to the **animatePath** function from Chapter 10, we refer the reader to Section 10.5 of this manual for a detailed explanation.

```
In[217]:= animateTree[G_Graph, T_List, opts___] := Module[{i, len},
  len = Length[T];
  Animate[HighlightGraph[G, T[[1 ;; i]], opts],
    {{i, 0, "step"}, 0, len, 1},
    AnimationRunning → False, AnimationRepetitions → 1]
]
```

```
In[218]:= animateTree[exercise3, primEdges[exercise3]]
```



Kruskal's Algorithm

Recall that Kruskal's algorithm, Algorithm 2 in Section 11.5, begins in the same way as Prim's algorithm, with the edge of smallest weight. The difference is that at each step, Kruskal's algorithm adds whatever edge is of least weight which does not create a simple circuit, regardless of whether it is incident to an edge already in the graph.

We begin with a function to test whether or not a given edge will create a simple circuit. Note that, during the steps of Kruskal's algorithm, we have a forest of trees. An edge will create a simple circuit if and only if both of its endpoints are in the same tree within the forest. We use the `ConnectedComponents` function to find the trees. The `ConnectedComponents` function returns a list of lists, where each inner list is the vertices within one of the connected components of the graph. We test an edge by looping through each of the connected components and making sure that both ends do not appear in the same component.

```
In[219]:= noCircuitQ[G_Graph, edge_] := Module[{components, C},
  components = ConnectedComponents[G];
  Catch[
    Do[If[MemberQ[C, edge[[1]]] &&
      MemberQ[C, edge[[2]]], Throw[False]]
    , {C, components}];
  Throw[True]
]
```

Now we implement Kruskal's algorithm. The function is as follows.

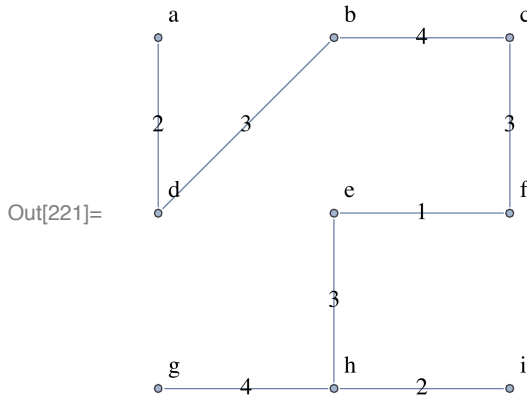
1. Initialize **edges** to the list of edges of the given graph and sort this list using the **edgeCompare** function created above.

2. Initialize **T** to the graph with all the vertices of **G** and no edges..
3. Consider the first edge in the **edges** list. Use **noCircuitQ** to determine if it is safe to add to the tree. If we can add it to the tree, add one or both of its ends as new vertices to **T** as needed and add the edge.
4. Regardless of whether **noCircuitQ** approves the addition of the first edge in **edges** to the tree, remove the edge from **edges** — either the edge is now used in the tree and so won't be used again, or its addition would create a circuit (a fact which won't change later).
5. Repeat steps 3 and 4 until $n - 1$ edges have been added, where n is the number of vertices.

```
In[220]:= kruskal[G_Graph, opts___] :=
  Module[{edges, T, n, i, newEdge, v},
    edges = Sort[EdgeList[G], edgeCompare[G]];
    T = Graph[VertexList[G], {}, opts];
    n = VertexCount[G];
    i = 1;
    While[i ≤ n - 1,
      newEdge = edges[[1]];
      If[noCircuitQ[T, newEdge],
        T = EdgeAdd[T, newEdge];
        PropertyValue[{T, newEdge}, EdgeWeight] =
          PropertyValue[{G, newEdge}, EdgeWeight];
        PropertyValue[{T, newEdge}, EdgeLabels] =
          PropertyValue[{G, newEdge}, EdgeWeight];
        i++
      ];
      edges = Delete[edges, 1]
    ];
    Do[PropertyValue[{T, v}, VertexCoordinates] =
      PropertyValue[{G, v}, VertexCoordinates];
      PropertyValue[{T, v}, VertexLabels] =
        PropertyValue[{G, v}, VertexLabels]
    , {v, VertexList[T]}];
  T
]
```

Note that Kruskal's algorithm produces the same minimum spanning tree for Exercise 3 as did our implementation of Prim's algorithm (in fact, this graph has a unique minimum spanning tree).

```
In[221]:= kruskalExercise3 = kruskal[exercise3, ImagePadding -> 10]
```



We can produce an animation, like we did for Prim's algorithm, by modifying the function to produce the list of edges in the order they are added and then using the **animateTree** function once again.

```
In[222]:= kruskalEdges[G_Graph] :=  

Module[{edges, edgeList, T, n, i, newEdge, v},  

edges = Sort[EdgeList[G], edgeCompare[G]];  

edgeList = {};  

T = Graph[VertexList[G], {}];  

n = VertexCount[G];  

i = 1;  

While[i ≤ n - 1,  

newEdge = edges[[1]];  

If[noCircuitQ[T, newEdge],  

T = EdgeAdd[T, newEdge];  

AppendTo[edgeList, newEdge];  

i++  

];  

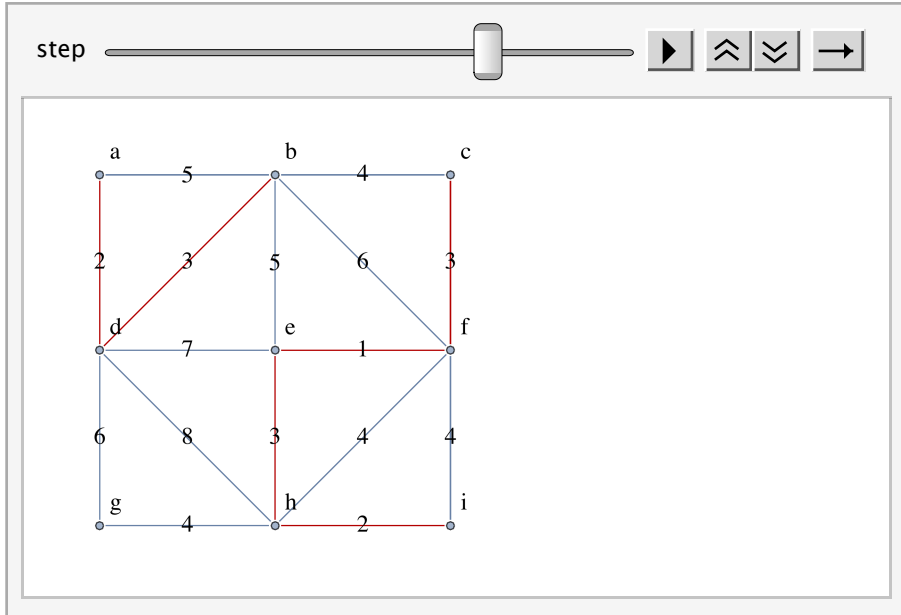
edges = Delete[edges, 1]  

];  

edgeList  

]
```

```
In[223]:= animateTree[exercise3, kruskalEdges[exercise3]]
```



By comparing the two animations, you can see how the two algorithms provide different routes to the same end result.

Solutions to Computer Projects and Computations and Explorations

Computer Projects 6

Given the ordered list of edges of an ordered rooted tree, find the universal addresses of its vertices.

Solution: Recall that the universal address of a vertex in an ordered rooted tree is defined as follows. The root has address 0 and its children have addresses 1, 2, 3, etc., in order. The address of every other vertex is defined recursively as $p.n$ where p is the address of the vertex's parent and n is 1 if the vertex is the first child of its parent, 2 if it is the second child, etc.

Before solving this problem, we will make the following assumption on the input: the edges are sorted according to the lexicographical order of the universal address of their terminal vertex. That is to say, the edges are listed in the order of their appearance from left to right and top to bottom when the tree is drawn in the usual way.

We build the ordered rooted tree (satisfying **orderedRootedTree0**) determined by the list of edges and add a “univ-address” property to each vertex containing the vertex's universal address. First, here is an ordered list of edges for an ordered rooted tree.

```
In[224]:= cp6Example = {"D" → "E", "D" → "C", "D" → "I", "E" → "L", "E" → "F",
  "I" → "B", "I" → "K", "I" → "H", "F" → "J", "F" → "G", "F" → "A"}

Out[224]= {D → E, D → C, D → I, E → L, E → F,
  I → B, I → K, I → H, F → J, F → G, F → A}
```

We have purposefully chosen vertex names that are out of order with respect to the ordering of the edges so that our construction is sure to rely only on the ordering of the edges and not the vertex labels.

Note that creating a graph with these edges only requires passing it to the Graph function.

```
In[225]:= cp6Graph = Graph[cp6Example];
```

Our first task is to turn this into an ordered rooted tree. Since we provided directed edges, it is already a rooted tree.

```
In[226]:= rootedTreeQ[cp6Graph]
```

```
Out[226]= True
```

To make this graph an ordered rooted tree, we need to set the “order” property for each vertex. For the root, we just set the root’s “order” to 0.

```
In[227]:= PropertyValue[{cp6Graph, findRoot[cp6Graph]}, "order"] = 0
```

```
Out[227]= 0
```

For the rest of the vertices, we need to do some more work. Our approach will be as follows. Loop through all of the edges in the original edge list, in order, keeping track of two variables, **curParent**, the current parent, and the **childOrder**. The **curParent** will initially be set to the root of the tree and **childOrder** will be initialized to 1. For the first edge in the list, we assign the child vertex (the terminal vertex of the ordered edge) “order” equal to **childOrder**. We then go to the next edge. If the **curParent** is the same as the parent vertex in this edge, then **childOrder** is incremented and we assign the child vertex of this edge an order equal to the new **childOrder** value. Otherwise, the parent vertex of this new edge is different from **curParent**. This indicates that we have moved on to a new parent with a new set of children, so we set **curParent** to this new parent and reset **childOrder** to 1. Here is the code for this step in the process.

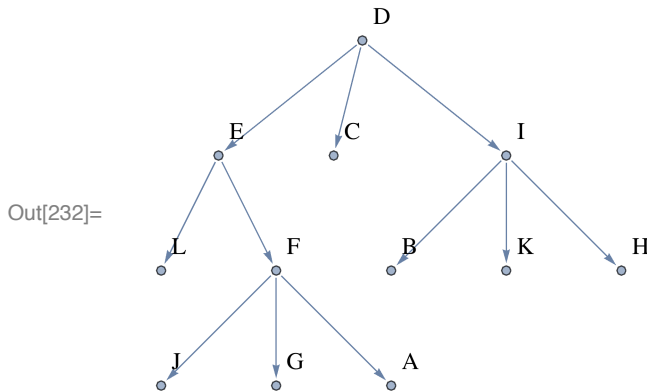
```
In[228]:= curParent = findRoot[cp6Graph];
childOrder = 1;
Do[If[thisEdge[[1]] ≠ curParent,
  curParent = thisEdge[[1]];
  childOrder = 1];
PropertyValue[{cp6Graph, thisEdge[[2]]}, "order"] =
  childOrder;
childOrder++
, {thisEdge, cp6Example}]
```

Now that the order attributes are set, our graph is an ordered rooted tree.

```
In[231]:= orderedRootedTreeQ[cp6Graph]
```

```
Out[231]= True
```

```
In[232]:= plotORTree[cp6Graph, VertexLabels → "Name", ImagePadding → 10]
```



We use the approach above to define a function.

```
In[233]:= treeFromList[L_List, opts___] :=
Module[{T, e, curParent, childOrder, thisEdge},
  T = Graph[L, opts];
  PropertyValue[{T, findRoot[T]}, "order"] = 0;
  curParent = findRoot[T];
  childOrder = 1;
  Do[If[thisEdge[[1]] ≠ curParent,
    curParent = thisEdge[[1]];
    childOrder = 1
  ];
  PropertyValue[{T, thisEdge[[2]]}, "order"] = childOrder;
  childOrder++
  , {thisEdge, L}];
T
]
```

We could now create a function that uses the pattern of a breadth first search to assign the universal addresses. But instead we will take this opportunity to use the built-in function BreadthFirstScan instead. The first two arguments will be the graph and the root of the graph, which explicitly tells *Mathematica* where to begin the breadth first search. The third argument will be a list specifying a function to use when the event “**DiscoverVertex**” occurs. The function is applied to two arguments, #1 refers to the vertex being discovered, and #2 refers to the vertex from which it is discovered, that is, its parent. The body of our function will check if the parent has “order” property 0, indicating that the discovered vertex is either the root or a child of the root. In this case, the “univ-address” property will be set to the vertex’s order. Otherwise, the “univ-address” will be set to the concatenation of the parent’s address and the vertex’s order.

```

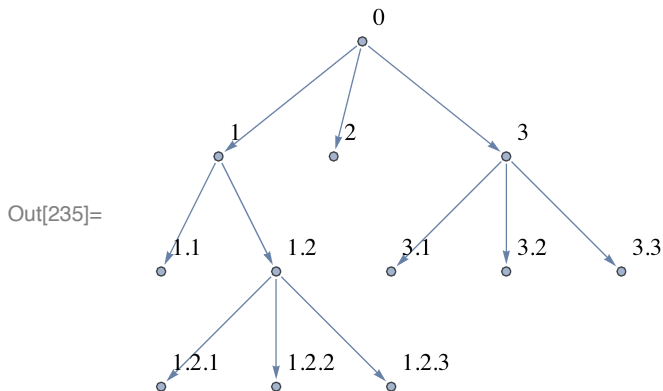
In[234]:= universalAddress[edgeList_List, opts___] := Module[{G},
  G = treeFromList[edgeList, opts];
  BreadthFirstScan[G, findRoot[G], {"DiscoverVertex" →
    ((If[PropertyValue[{G, #2}, "order"] == 0,
      (* root or root's child *)
      PropertyValue[{G, #1}, "univ-address"] =
        ToString[PropertyValue[{G, #1}, "order"]],
      (* descendant of a root's child *)
      PropertyValue[{G, #1}, "univ-address"] =
        PropertyValue[{G, #2}, "univ-address"] <> "." <>
        ToString[PropertyValue[{G, #1}, "order"]]]];
    (* set address as vertex label *)
    PropertyValue[{G, #1}, VertexLabels] =
      PropertyValue[{G, #1}, "univ-address"]
    ) & )]];
  G
]

```

```

In[235]:= universalAddress[cp6Example, ImagePadding → 10]

```



Computations and Explorations 1

Display all trees with six vertices.

Solution: To solve this problem, we make use of a recursive definition of trees. The empty graph is a tree, the graph with a single vertex is also a tree, and the graph with two vertices with an edge between them is a tree. Given any tree, we can form a new tree with one additional vertex by adding the new vertex as a leaf connected to any one of the original vertices. (The reader can verify that this indeed creates all trees with one more vertex.)

We shall create a function, called **extendTrees**, that accepts as input a list of trees on n vertices and returns the resulting list of trees on $n + 1$ vertices. For each of the original trees, we consider each vertex of the tree and create a new tree by adding a leaf to it.

```

In[236]:= extendTrees[trees_List] := Module[{newTrees, newV, T, v, newT},
  newTrees = {};
  newV = VertexCount[trees[[1]]] + 1;
  Do[
    Do[
      newT = VertexAdd[T, newV];
      newT = EdgeAdd[newT, v → newV];
      AppendTo[newTrees, newT]
      , {v, VertexList[T]}]
    , {T, trees}];
  newTrees
]

```

We can now use this procedure to determine all trees on four vertices. Finding all the trees of larger sizes is left to the reader.

```

In[237]:= allTrees = {Graph[{1, 2}, {1 → 2}, DirectedEdges → True]}

```

```

Out[237]= { • ←————— • }

```

```

In[238]:= For[i = 3, i ≤ 4, i++, allTrees = extendTrees[allTrees]];

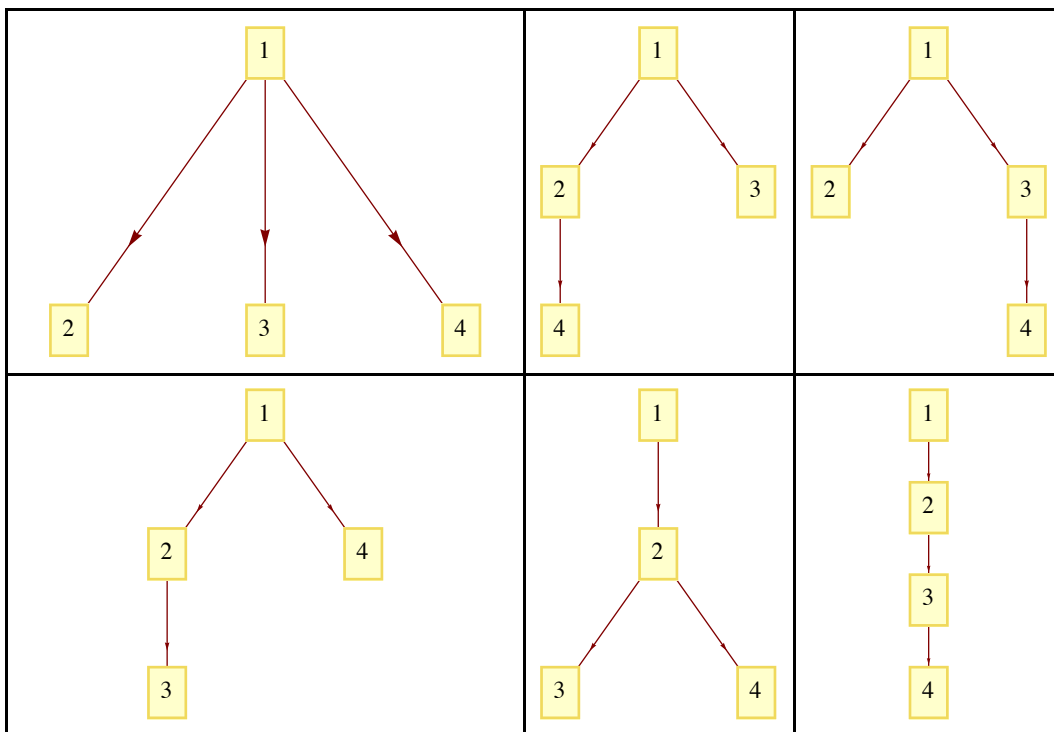
```

We now display the trees on four vertices. We Map **rootedPlot** over the list in order to display the trees properly.

```

In[239]:= Grid[Partition[Map[rootedPlot[#, VertexLabeling → True] &,
  allTrees], 3], Frame → All]

```



Computations and Explorations 3

Construct a Huffman code for the symbols with ASCII codes given the frequency of their occurrence in representative input.

Solution: ASCII, which stands for American Standard Code for Information Interchange, includes 128 characters, including 33 non-printing characters. Most of the non-printing characters, with the exception of the space and the carriage return or newline character, however, are rarely used. We will focus on the standard characters of English and the newline character.

Since we've already created the function, **huffmanCode**, which creates a Huffman code based on a list of character/weight pairs, the main work we need to do is determine the frequencies of characters in a sample input. We use the following input, which contains letters, punctuation, and newline characters.

```
In[240]:= inputText = "The quick brown fox said,
\"How do you do, my friend?\"
Then he ran very quickly off into the sunset."
```

```
Out[240]= The quick brown fox said,
" How do you do, my friend?"
Then he ran very quickly off into the sunset.
```

We can turn this string into a list by applying the Characters function.

```
In[241]:= textList = Characters[inputText]

Out[241]= {T, h, e,  , q, u, i, c, k,  , b,
           r, o, w, n,  , f, o, x,  , s, a, i, d,  ,
           , " , H, o, w,  , d, o,  , y, o, u,  , d, o,
           , ,  , m, y,  , f, r, i, e, n, d, ? , " ,
           , T, h, e, n,  , h, e,  , r, a, n,  , v, e, r, y,  , q, u, i, c, k,
           l, y,  , o, f, f,  , i, n, t, o,  , t, h, e,  , s, u, n, s, e, t, . }
```

Using FullForm, we can have *Mathematica* display the newline characters for us explicitly. Also, *Mathematica* differentiates between a newline character and a space character.

```
In[242]:= textList[[26]] // FullForm
```

```
Out[242]//FullForm=
"\n"
```

```
In[243]:= textList[[4]] // FullForm
```

```
Out[243]//FullForm=
" "
```

```
In[244]:= textList[[26]] == textList[[4]]
```

```
Out[244]= False
```

We will calculate the frequencies of the characters in our text by applying the Tally function. When applied to a list, Tally returns a list of pairs consisting of the unique elements of the original list and the number of times each appears.

```
In[245]:= Tally[Characters[inputText]]
```

```
Out[245]= {{T, 2}, {h, 4}, {e, 7}, { , 17}, {q, 2}, {u, 4},
           {i, 5}, {c, 2}, {k, 2}, {b, 1}, {r, 4}, {o, 8}, {w, 2},
           {n, 6}, {f, 4}, {x, 1}, {s, 3}, {a, 2}, {d, 4}, {,, 2}, {
           , 2}, {" , 2}, {H, 1}, {y, 4}, {m, 1},
           {?, 1}, {v, 1}, {l, 1}, {t, 3}, {., 1}}
```

The Huffman function expects the input to consist of the characters and their relative frequency. So we need to divide the counts from the Tally output by the number of characters.

```
In[246]:= Length[Characters[inputText]]
```

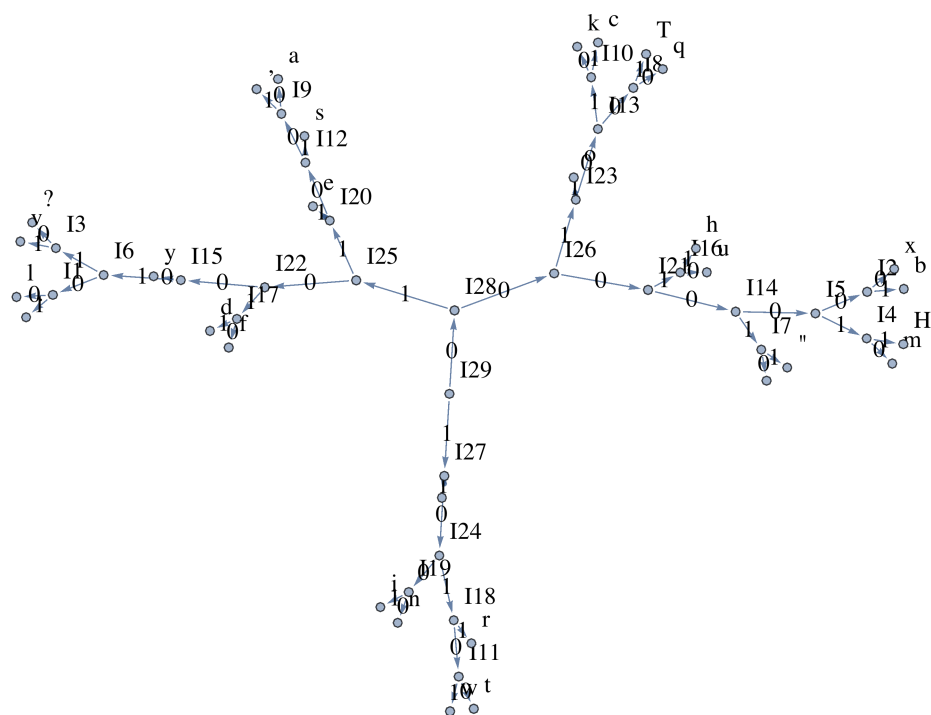
```
Out[246]= 99
```

```
In[247]:= foxList = Tally[Characters[inputText]] /. {c_, n_} -> {c, n/99}
```

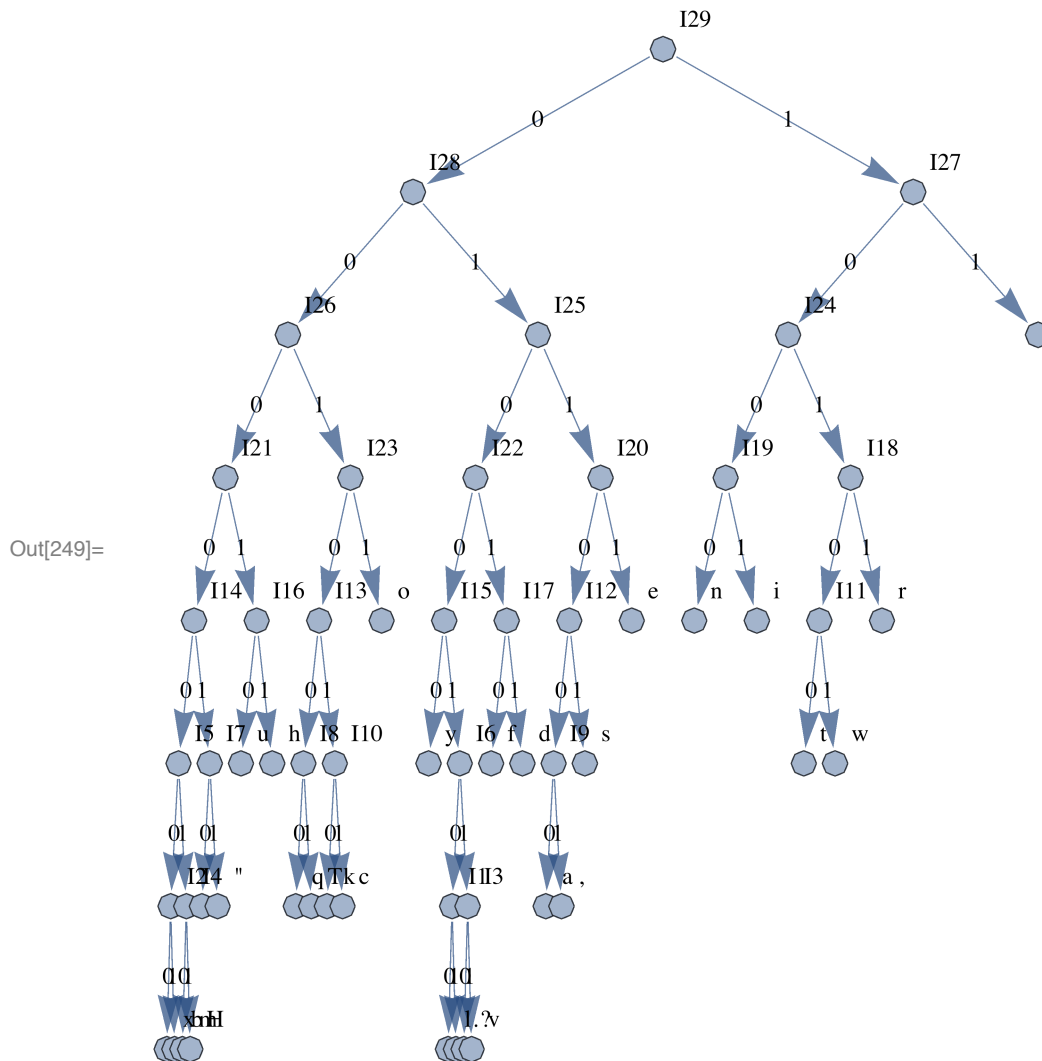
```
Out[247]= {{T,  $\frac{2}{99}$ }, {h,  $\frac{4}{99}$ }, {e,  $\frac{7}{99}$ }, { ,  $\frac{17}{99}$ }, {q,  $\frac{2}{99}$ }, {u,  $\frac{4}{99}$ }, {i,  $\frac{5}{99}$ },
           {c,  $\frac{2}{99}$ }, {k,  $\frac{2}{99}$ }, {b,  $\frac{1}{99}$ }, {r,  $\frac{4}{99}$ }, {o,  $\frac{8}{99}$ }, {w,  $\frac{2}{99}$ }, {n,  $\frac{2}{33}$ },
           {f,  $\frac{4}{99}$ }, {x,  $\frac{1}{99}$ }, {s,  $\frac{1}{33}$ }, {a,  $\frac{2}{99}$ }, {d,  $\frac{4}{99}$ }, {,,  $\frac{2}{99}$ }, {
           ,  $\frac{2}{99}$ }, {" ,  $\frac{2}{99}$ }, {H,  $\frac{1}{99}$ }, {y,  $\frac{4}{99}$ }, {m,  $\frac{1}{99}$ },
           {?,  $\frac{1}{99}$ }, {v,  $\frac{1}{99}$ }, {l,  $\frac{1}{99}$ }, {t,  $\frac{1}{33}$ }, {.,  $\frac{1}{99}$ }}
```

Finally, we're able to apply the **huffmanCode** function.

```
In[248]:= foxCode =  
          huffmanCode[foxList, VertexLabels → "Name", ImagePadding → 5]
```



```
In[249]:= drawBinaryTree[foxCode]
```



You will need to enlarge the image substantially in order to see the result in a readable format.

Computations and Explorations 8

Draw the complete game tree for a game of checkers on a 4×4 board.

Solution: We will provide a partial solution to this problem; the reader is left to complete the full solution. Specifically, we will create a *Mathematica* function called **movePiece** that will determine all possible new checker arrangements given the current state of the board and the player whose turn it is. Once this function is created, the reader must determine how to represent these board positions as vertices and edges, how to determine the next level of the game tree, as well as the halting conditions.

Before writing this function, however, we must establish a representation of the board. Naturally, we will use a matrix whose size is the size of the board. Empty board spaces will contain 0. Board spaces in which a regular white or black piece is sitting will be represented by 1 or 2, respectively. Kings will

be represented by negative values, -1 for a white king and -2 for a black king. The following represents an initial board before any moves have been made.

```
In[250]:= checkersStart =
           {{0, 2, 0, 2}, {0, 0, 0, 0}, {0, 0, 0, 0}, {1, 0, 1, 0}};
           checkersStart // MatrixForm
```

Out[251]//MatrixForm=

$$\begin{pmatrix} 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Given a matrix representing a board state and an integer representing which side's turn it is, the function **movePiece** will list all of the possible results of the player's move. It operates as follows:

1. Initialize **newBoards**, which will be the list of all possible boards that result from the current move, to the empty list.
2. If **side** is 1, then normal pieces move up the board from bottom to top and we set **direction** to -1, since the index of rows in a matrix decrease as we move up the board. If the **side** is 2, then **direction** is set to +1.
3. Begin a pair of **For** loops, with indices **r** and **c**. These **For** loops allow us to consider each possible board location. In each position, we want to know if that location holds a piece belonging to the current player. If it is 1's turn, then that player's normal pieces are represented by a 1 in the position and the player's kings are represented by a -1 in the position. Likewise, 2's pieces are represent by 2 or by -2. Thus we can determine if a position holds a player's piece by comparing the absolute value of the matrix entry with the **side**. If the square does not hold a piece belonging to the current player, we simply move on to the next location.
4. Check to see if the piece is a king and set the variable **isKing** to 1 if it is a king or 0 if not. We then begin a **For** loop from 0 to the value of **isKing**. If **isKing** is 0, the loop executes only once. If **isKing** is 1, then the loop will execute twice. The index of this loop, **king**, is used to control **rowDir**, the current direction being considered. **rowDir** is either the same as **direction** or, in the case of the second iteration for a king, the reverse direction.
5. We now check to see if the possible moves keep the piece on the board. First, we make sure that **r+rowDir**, that is, the row in which the piece would move to, is still between 1 and 4 (the possible rows).
6. Assuming moving the piece won't take it off the top or bottom of the board, consider the left and right moves. We do this with a **For** loop which sets the variable **colDir** to -1 and then to +1. Again, we check to see that **c+colDir**, the current column plus the proposed change to the column position, is still on the board.
7. At this point we know that the board position (**r+rowDir, c+colDir**) is actually a board position. There are now three possibilities: the position is empty, there is an enemy piece in the square, there is a friendly piece in the square.
8. In the first case, the position is empty, we want to move the piece to that location. We make a copy of the **board** matrix (so we don't modify the original board). Then we make the move and add the new board to the list. Note that we also check to see if the piece becomes a king by moving into this position.

9. In the second case, there is an enemy in the square, then we test to see if it is possible to jump. That is, we must make sure that the landing location after the jump is both on the board and empty. If so, we make the jump, i.e., we copy the **board** and make the necessary modifications. If not, then the move is not possible.

10. In the third case, the move is not possible and we do nothing.

Here is the function.

```
In[252]:= movePiece[board_List, side_Integer] :=
Module[{newBoards = {}, direction, r,
  c, isKing, king, rowDir, colDir, newB},
  direction = Switch[side, 1, -1, 2, 1];
  For[r = 1, r ≤ 4, r++,
    For[c = 1, c ≤ 4, c++,
      If[Abs[board[[r, c]]] == side,
        If[board[[r, c]] < 0, isKing = 1, isKing = 0];
        For[king = 0, king ≤ isKing, king++,
          rowDir = Switch[king, 0, direction, 1, -direction];
          If[r + rowDir ≥ 1 && r + rowDir ≤ 4,
            For[colDir = -1, colDir ≤ 1, colDir = colDir + 2,
              If[c + colDir ≥ 1 && c + colDir ≤ 4,
                Which[board[[r + rowDir, c + colDir]] == 0,
                  newB = board;
                  If[(r + rowDir == 1 && side == 1) ||
                    (r + rowDir == 4 && side == 2)) && board[[r, c]] > 0,
                    newB[[r + rowDir, c + colDir]] = -1 * board[[r, c]],
                    newB[[r + rowDir, c + colDir]] = board[[r, c]]
                  ];
                  newB[[r, c]] = 0;
                  AppendTo[newBoards, newB],
                Abs[board[[r + rowDir, c + colDir]]] ≠ side,
                If[r + 2 * rowDir ≥ 1 && r + 2 * rowDir ≤ 4 &&
                  c + 2 * colDir ≥ 1 && c + 2 * colDir ≤ 4,
                  If[board[[r + 2 * rowDir, c + 2 * colDir]] == 0,
                    newB = board;
                    If[(r + 2 * rowDir == 1 && side == 1) || (r + 2 * rowDir ==
                      4 && side == 2)) && board[[r, c]] > 0, newB[[r +
                      2 * rowDir, c + 2 * colDir]] = -board[[r, c]],
                    newB[[r + 2 * rowDir, c + 2 * colDir]] = board[[r, c]]
                    ];
                    newB[[r, c]] = 0;
                    newB[[r + rowDir, c + colDir]] = 0;
                    AppendTo[newBoards, newB]
                  ]
                ]
            ]
          ]
        ]
      ]
    ]
  ]
```

```

    ]
  ]
]
];
newBoards
]

```

We now demonstrate a few steps using the function.

```

In[253]:= move1 = movePiece[checkersStart, 1];
MatrixForm /@ move1

```

Out[254]= $\left\{ \begin{pmatrix} 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \right\}$

```

In[255]:= move2 = movePiece[move1[[1]], 2];
MatrixForm /@ move2

```

Out[256]= $\left\{ \begin{pmatrix} 0 & 0 & 0 & 2 \\ 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \right\}$

```

In[257]:= move3 = movePiece[move2[[3]], 1];
MatrixForm /@ move3

```

Out[258]= $\left\{ \begin{pmatrix} 0 & 2 & 0 & 0 \\ 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 2 & 0 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix} \right\}$

```

In[259]:= move4 = movePiece[move3[[2]], 2];
MatrixForm /@ move4

```

Out[260]= $\left\{ \begin{pmatrix} 0 & 0 & 0 & -1 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \right\}$

Exercises

1. Write *Mathematica* functions for finding the eccentricity of a vertex in an unrooted tree and for finding the center of an unrooted tree. (Eccentricity and center are defined in prelude to Exercise 39 of Section 11.1 of the text.)
2. Develop a *Mathematica* function for constructing rooted Fibonacci trees. (See the prelude to Exercise 45 of Section 11.1 for a definition of a Fibonacci tree.)
3. Develop a *Mathematica* function for listing the vertices of an ordered rooted tree in level order.
4. Compare the performance of binary search trees to linear search as follows:
 - a. Write a function, **linearSearch**, that takes two inputs, a list of integers and an integer to find, and checks each element of the list in order until the input is found, at which time it returns True. If the desired integer is not found, it is added the end of the list.
 - b. Use the function **RandomSample[Range[n], n]** for a positive integer n to create a list of the first n integers in random order, with an appropriately large n . Apply the **makeBST** function to the list to create a binary search tree for the data.
 - c. Randomly select some positive integers to search for. The RandomInteger function could be useful here.
 - d. Use both **linearSearch** and **binaryInsertion** to find the integers from part c in the list and tree, respectively. Time them using the typical Timing function. Repeat this for 100 different initial lists of integers and compare the resulting times. Compare these data (representing average-case complexity) with the theoretical worst-case results of n comparisons for **linearSearch** and $\log(n)$ for **binaryInsertion**.
5. Construct a *Mathematica* function for decoding a message which was encoded with a Huffman code. That is, given a Huffman coding tree produced by the HuffmanCode procedure and a message encoded by the **encodeString** function, the algorithm should return the original string.
6. Use the Shakespearean sonnets to estimate the frequency of characters used by Shakespeare. (See Section 7.3 of this manual to see how to read the data into *Mathematica* and make use of the functions given in the solution to Computations and Explorations 3 above to compute the frequencies of characters used in the poems.) Then create a Huffman code based on the sonnets and encode the ShakespeareData.txt with the Huffman code. Compare the storage space required by the Huffman encoded version of the file as opposed to the space that would be used to encode the file in ASCII format, assuming each ASCII character requires 7 bits.
7. Construct an undirected weighted graph which has at least two different minimum spanning trees and for which the **prim** and **kruskal** algorithms will return different results.
8. Write a *Mathematica* function implementing the reverse-delete algorithm for constructing minimal spanning trees. (The reverse-delete algorithm is described in the prelude to Exercise 34 in Section 11.5.)
9. Explore the relative complexity of **prim**, **kruskal**, and the reverse-delete function you created in the previous exercise. Use the RandomGraph function to experiment with their performance. The expression **RandomGraph[{v, e}]** will produce a random weighted connected graph with v vertices and e edges. You will then need to add edge weights with a function you design yourself. For each algorithm, can you find properties that you can impose

on the graphs that will ensure that the algorithm will outperform the others?

10. Develop a *Mathematica* function for producing degree-constrained spanning trees, which are defined in the Supplementary Exercises for Chapter 11. Use this function on a set of randomly generated graphs to attempt to construct degree-constrained spanning trees in which each vertex has degree no larger than 3.
11. Use *Mathematica* to analyze the game of Nim with different starting conditions via the technique of game trees. (See Example 6 in Section 11.2 for a description of the game of Nim.)
12. Use *Mathematica* to analyze the game of checkers on square boards of different sizes via the technique of game trees. (See the solution to Computations and Explorations 8 for the beginnings of a solution.)
13. Develop *Mathematica* functions for finding a path through a maze using the technique of backtracking.
14. Develop *Mathematica* functions for solving Sudoku puzzles using the technique of backtracking.
15. Use *Mathematica* to generate as many graceful trees as possible. (See the Supplementary Exercises of Chapter 11 for a definition of graceful.) Based on the examples you find, make conjectures about graceful trees.
16. Alter the postfix expression evaluator, **evalPostfix**, to handle prefix expressions.