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 Maple. 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 see how to use Maple 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 Maple 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.

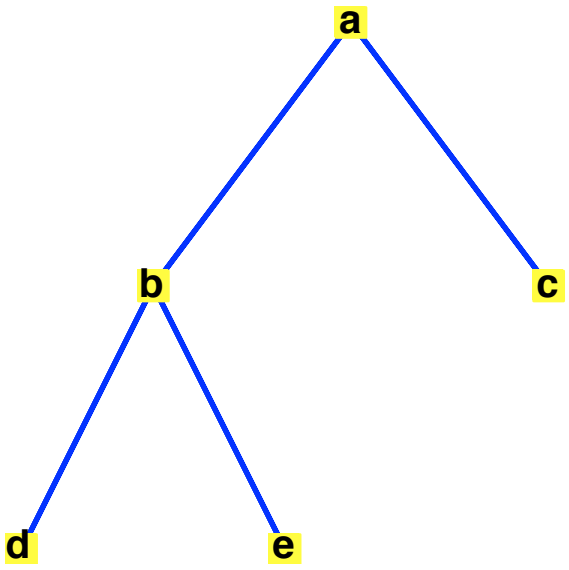
Unrooted Trees

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 just create a graph as we did in the previous chapter. We begin by loading the graph theory package and then creating a simple tree with the `Graph` command.

```
> with(GraphTheory):
> firstTree:= Graph({{"a","b"},{"a","c"},{"b","d"},{"b","e"}});
```

firstTree := Graph 1: an undirected unweighted graph with 5 vertices and 4 edge(s) **(11.1)**

```
> DrawGraph(firstTree);
```



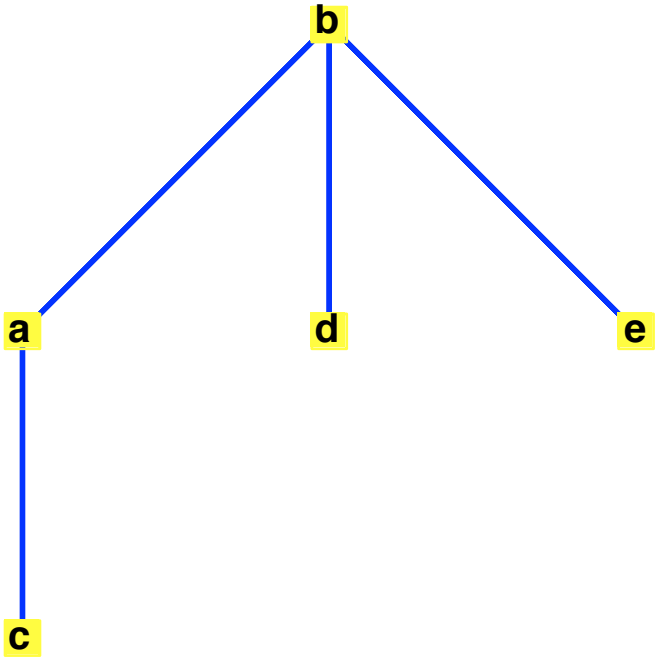
The first thing you may notice is that Maple has automatically drawn this in the traditional way, which tells us that Maple recognized the graph as a tree. The **IsTree** command can be used to check if an undirected graph is a tree.

```
> IsTree(firstTree);
```

true **(11.2)**

Recall from Section 10.1 of this manual that the **DrawGraph** command can take the optional **style=tree** argument in order to make sure Maple draws the graph as a tree. The **DrawGraph** command, when used on trees, can also take an optional argument of the form **root=r** to specify which vertex should be drawn as the root.

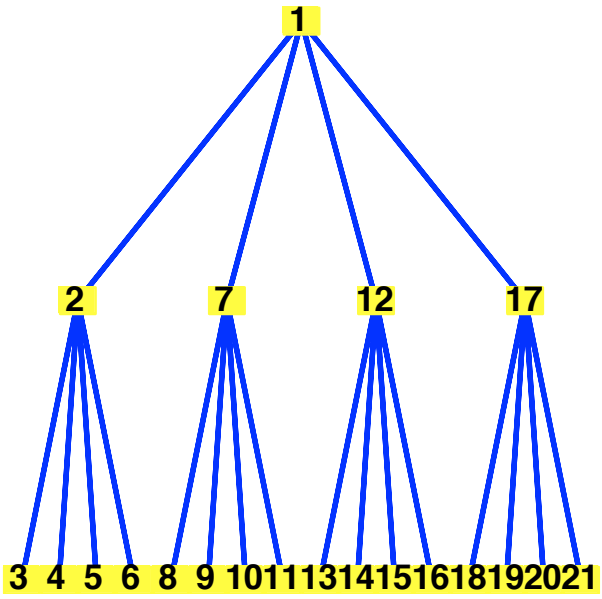
```
> DrawGraph(firstTree,style=tree,root="b");
```



We make three comments about the **root=r** argument. First, it can only be used when the **style=tree** option is explicitly given. Second, as of Maple 14, this is irreversible and unchangeable — issuing the command with a different vertex will not give the expected result. Third, while this option makes the tree appear to have the specified root, it does not make it a rooted tree, in the sense used by the textbook. That is to say, other than the positions of the vertices, there is no information stored in the data of **firstTree** to indicate that vertex *b* is the root or even that vertex *c* is a child rather than a parent of *a*.

Maple also provides commands 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. (Maple uses the term depth, which is synonymous with height.) The command **CompleteKaryTree** in the **SpecialGraphs** subpackage produces the unrooted version of the complete *k*-ary directed tree of height *h*, where *k* and *h* are given as the two arguments to the command.

```
> DrawGraph(SpecialGraphs[CompleteKaryTree](4,2));
```



The `CompleteBinaryTree` command can be used in the case $k = 2$.

Before moving on to rooted trees, note that Maple also provides a command `IsForest` for checking whether a graph is a forest (*i.e.*, a collection of trees). The only argument is the graph.

Rooted Trees

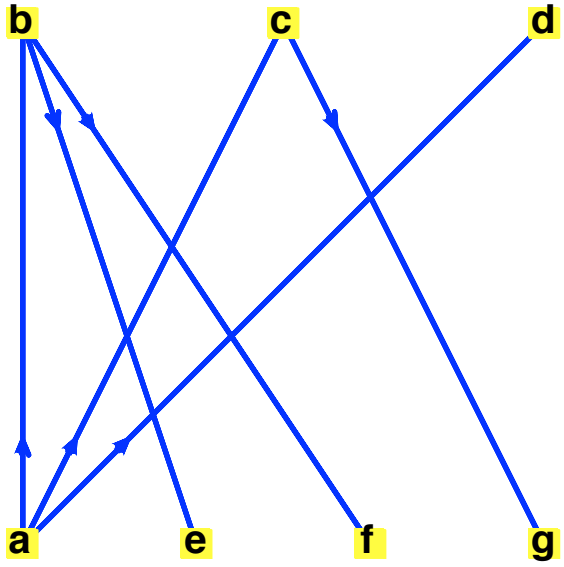
Next we consider rooted trees. Recall that a rooted tree is a directed graph whose underlying undirected graph is a tree and in which one vertex is designated as the root and all edges are directed away from the root. For example, the following graph is a rooted tree.

```
> firstRooted := Graph({["a","b"],["a","c"],["a","d"],["b",
  "e"],["b","f"],["c","g"]});
```

firstRooted := Graph 2: a directed unweighted graph with 7 vertices and 6 arc(s) **(11.3)**

If we ask Maple to draw this graph with `DrawGraph`, however, it will not draw it in the form of a tree like it did with `firstTree`. This is because Maple does not recognize this graph as a tree.

```
> DrawGraph(firstRooted);
```



```
> IsTree(firstRooted);
Error, (in GraphTheory:-IsTree) graph must be undirected
```

While Maple provides some support for unrooted trees, its existing packages are not equipped to recognize and do computations with rooted trees. Much of the remainder of this section will be devoted to filling this gap in Maple's functionality. This will provide you with the tools you need to better explore trees in the remainder of the chapter.

A tree type

In Chapter 9 we discussed Maple types. It will once again be useful to declare types for our trees. This will enable automatic type checking for our procedures, which will help prevent attempts to use procedures on objects that are not trees. Moreover, the declaration of types for trees will formalize what it means to be a tree object in Maple.

A rooted tree, by definition, has a root. It will be convenient to store that information with the tree by setting a graph attribute that indicates which vertex is the root. This will be useful, in that it will free us from having to list the root as an argument to every procedure we create. It is also good programming practice — the root of a rooted tree is information that the tree should "know about itself," that is to say, the tree should include the root as part of its data.

We begin by creating a type for unrooted trees before returning to the rooted situation. We define the type by creating a procedure which takes one argument, the potential tree, and returns true or false depending on whether it is or is not actually a tree. For unrooted trees, we will be calling the **IsTree** command to do the bulk of the work for us.

We will also use the **try** statement in this type definition. A **try** statement is the primary method for catching and handling errors. We saw above that applying the **IsTree** command to a directed graph produces an error. In the type procedure below, we use the **try** statement as follows. First is the **try** keyword followed by the code that could potentially raise an error. In this case, it is the **IsTree** command that could raise an error. After the error-prone code, we use the **catch:** line. If anything in the code in the **try** block raises an error, the code following the **catch:** keyword is executed to "handle" the error. In this type definition, we handle any errors by setting the return value to false. (Note: the **try** statement structure is very flexible. Refer to the Maple help pages for more detail.)

```
> `type/Tree` := proc(obj)
    local result;
    try
      result := GraphTheory[IsTree](obj);
    catch:
      result := false;
    end try;
    return result;
  end proc:
```

We can explicitly check to see if an object is of a specified type by using the **type** command with the object and the name of the type as arguments.

```
> type(firstTree,Tree);
```

true **(11.4)**

```
> type(firstRooted,Tree);
```

false **(11.5)**

Now we'll create the type **RTree** for rooted trees. As discussed earlier, we insist that an object of

this type stores its root as a graph attribute. To set an attribute, we use the **SetGraphAttribute** command with two arguments: the graph, and the attribute and its value in **tag=value** format. We use the tag "root" and the value will be the name of the root.

```
> SetGraphAttribute(firstRooted,"root"="a");
```

The **GetGraphAttribute** command with the name of the graph and the tag, in this case "root", will return the value of the tag.

```
> GetGraphAttribute(firstRooted,"root");
                                "a"                                (11.6)
```

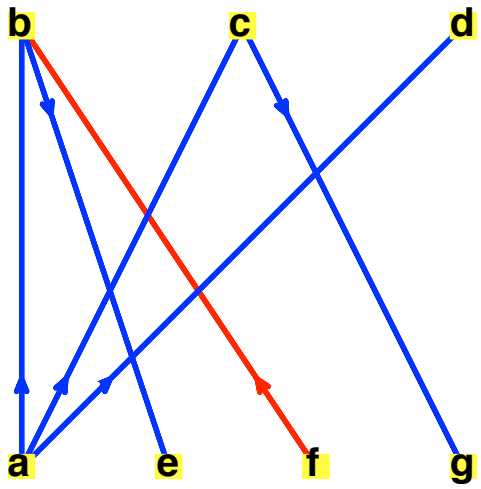
As with the **Tree** type, we will define the type by creating a procedure that returns true for rooted trees and false for all other objects. We will need to check three things: first, that the "root" graph attribute has been set; second, that the underlying undirected graph is a tree; and third, that all edges point away from the root. For the first part, we'll just check that **GetGraphAttribute** returns a name of a vertex for the tag "root". Note that if a tag is not set, **GetGraphAttribute** will return **FAIL**.

For the second part, we can simply combine **IsTree** with the **UnderlyingGraph** command, which takes a directed graph and returns the underlying undirected graph. For example, we apply this to our example above.

```
> IsTree(UnderlyingGraph(firstRooted));
                                true                               (11.7)
```

The third part of the test is that every edge is directed away from the root. Before we implement the test, consider the following example, which is identical to **firstRooted** except the edge $[a, e]$ has been reversed.

```
> notRooted := Graph({["a","b"],["a","c"],["a","d"],["b","e"],
  ["f","b"],["c","g"]});
   notRooted := Graph 3: a directed unweighted graph with 7 vertices and 6 arc(s)   (11.8)
> HighlightEdges(notRooted,{["f","b"]},red);
> DrawGraph(notRooted);
```




```
> IsTree(UnderlyingGraph(notRooted));
                                true                               (11.9)
```

It is easy to see that the edge $[e, a]$ violates the requirement that all edges are directed away from the

root. Checking this computationally, however, can be a bit tricky to do directly. Instead, we will take an indirect route. Instead of checking that all edges are directed away from the root, we'll check that all the vertices are accessible from the root. Since the underlying graph is a tree, we know that there are no circuits. Thus the only way for there to be a path from the root to a vertex is for all the edges to be directed away from root. Therefore, if all the vertices are reachable from the root, then all the edges are in the proper direction.

To implement this, we will build a list of vertices accessible from the root. This list will be initialized to contain the purported root. Then we add to the list all of the vertices which are terminal vertices of edges with the purported root as the initial vertex. (The **Departures** command, discussed in Section 10.3, is useful here.) Once that's done, the list of accessible vertices consists of the root and all of the children of the root. For the second element in the list, we add all of its children, *i.e.*, all the vertices accessible from it. Continuing in this fashion, for each element in the list of accessible vertices, we add its children to the list. When we reach the end of the list, it contains all of the vertices that can be reached from the root. If it has the same members as the list of all vertices in the graph, then the graph satisfies the second condition of having all edges directed away from the root.

We now put these three elements together to create the type.

```
> `type/RTree` := proc(obj)
    local R, Alist, v, i;
    uses GraphTheory;
    if not type(obj,Graph) then
       return false
    end if;
    if not IsTree(UnderlyingGraph(obj)) then
      return false;
    end if;
    R := GetGraphAttribute(obj,"root");
    if not R in Vertices(obj) then
      return false;
    end if;
    Alist := [R];
    i := 1;
    while i <= nops(Alist) do
      Alist := [op(Alist),op(Departures(obj,Alist[i]))];
      i := i + 1;
    end do;
    if {op(Alist)} = {op(Vertices(obj))} then
      return true;
    else
      return false;
    end if;
  end proc:
```

```
> type(firstRooted,RTree);
```

true **(11.10)**

```
> type(notRooted,RTree);
```

false **(11.11)**

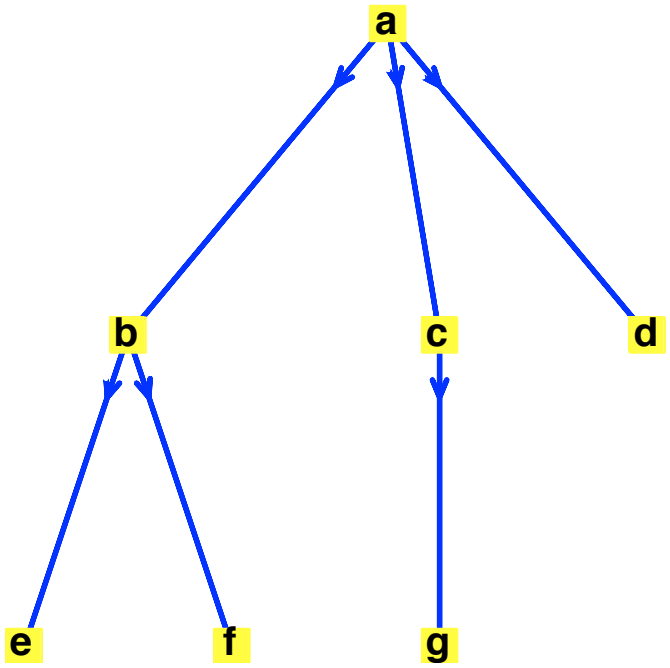
Drawing rooted trees

Now that we can test to see that a graph is in fact a rooted tree, let's get Maple to draw rooted trees so that they look like trees. Since the underlying graph of a rooted tree is itself a tree, we can

determine the best locations for drawing the vertices from the way Maple draws the underlying graph. To do this, we first apply **DrawGraph** to the underlying graph with the **style** and **root** options set but with the output suppressed. This is a necessary step since it is the **DrawGraph** command that causes Maple to calculate vertex positions. We then use the command **GetVertexPositions** on the underlying graph. And finally we use **SetVertexPositions** on the rooted tree and with the positions garnered from the underlying graph. (These commands were initially discussed in Section 10.2.)

```
> DrawRTree := proc(G::RTree)
    local R, U, P;
    uses GraphTheory;
    R := GetGraphAttribute(G,"root");
    U := UnderlyingGraph(G);
    DrawGraph(U,style=tree,root=R):
    P := GetVertexPositions(U,style=tree,root=R);
    SetVertexPositions(G,P);
    DrawGraph(G);
  end proc:
```

```
> DrawRTree(firstRooted);
```



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

We now consider commands related to identifying particular vertices in a rooted tree and relations between them.

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.

```
> IsParentOf := proc(T::RTree, p, c)
    return GraphTheory[HasArc](T,[p,c]);
  end proc:
```

```
> IsParentOf(firstRooted,"b","f");
```

true **(11.12)**

```
> IsParentOf(firstRooted,"b","d");
```

false **(11.13)**

Next, we consider the question of finding the parent of a given vertex. This can be done as an application of the **Arrivals** command, which returns the list of vertices that are initial vertices for the edges with the given vertex at the terminal end. Assuming the graph is in fact a rooted tree, there can be at most one such vertex. If the vertex is the root, there will be no parent and the procedure will return **FAIL**.

```
> FindParent := proc(T::RTree, v)
    local A;
    A := GraphTheory[Arrivals](T,v);
    if nops(A) = 1 then
      return op(A);
    elif nops(A) = 0 then
      return FAIL;
    else
      error "The given graph is not a tree.";
    end if;
  end proc:
> FindParent(firstRooted,"d");
```

"a" **(11.14)**

```
> FindParent(firstRooted,"root");
```

FAIL **(11.15)**

For the related question of determining all children of the given tree, we use the **Departures** command, which returns the list of vertices that are terminal vertices for the edges with the given vertex at the initial end.

```
> FindChildren := proc(T::RTree, v)
    return GraphTheory[Departures](T,v);
  end proc:
> FindChildren(firstRooted,"a");
```

["b", "c", "d"] **(11.16)**

```
> FindChildren(firstRooted,"f");
```

[ ] **(11.17)**

The **FindChildren** procedure also indicates how we can test a vertex to determine if it is an internal vertex or a leaf.

```
> IsInternal := proc(T::RTree, v)
    if GraphTheory[Departures](T,v) <> [] then
      return true;
    else
      return false;
    end if;
  end proc:
> IsLeaf := proc(T::RTree, v)
    if GraphTheory[Departures](T,v) = [] then
      return true;
    else
      return false;
    end if;
  end proc:
```

```
> IsInternal(firstRooted,"a");
```
true **(11.18)**

```
> IsLeaf(firstRooted,"a");
```
false **(11.19)**

```
> IsLeaf(firstRooted,"f");
```
true **(11.20)**

We can determine all the leaves of a given tree by testing each vertex with `IsLeaf`.

```
> FindLeaves := proc(T::RTree)
    local Leaves, v;
    uses GraphTheory;
    Leaves := {};
    for v in Vertices(T) do
      if IsLeaf(T,v) then
        Leaves := Leaves union {v};
      end if;
    end do;
    return Leaves;
  end proc:
> FindLeaves(firstRooted);
```
{"d", "e", "f", "g"} **(11.21)**

Ordered Rooted Trees

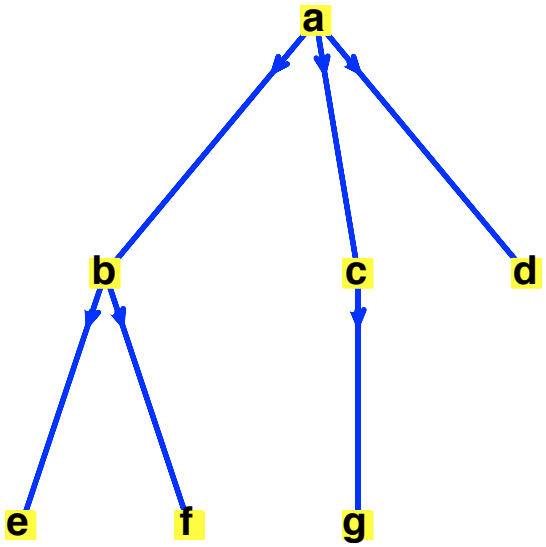
Recall that an ordered rooted tree is a rooted tree in which the children of each internal vertex are ordered.

Representing ordered rooted trees

To represent an ordered rooted tree in Maple, 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 duplicate `firstRooted` from above and make the duplicate ordered.

```
> firstOrdered := CopyGraph(firstRooted);
```
firstOrdered := Graph 4: a directed unweighted graph with 7 vertices and 6 arc(s) **(11.22)**

```
> DrawRTree(firstOrdered);
```



Maple 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, an "order" attribute. 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. Rather than manually using the **SetVertexAttribute** command for each vertex, we'll define a procedure.

The first argument to this procedure 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 procedure will loop through the vertices of the tree and set the "order" vertex attributes.

```
> SetChildOrder := proc(T::RTree, orderL::list(nonnegint))
    local V, i;
    uses GraphTheory;
    V := Vertices(T);
    if nops(V) <> nops(orderL) then
      error "List must have one entry per vertex.";
    end if;
    for i from 1 to nops(V) do
      SetVertexAttribute(T,V[i],"order"=orderL[i]);
    end do;
  end proc:
```

Since the "order" values in the list given to the procedure must match the order of the vertices that is returned from `Vertices(T)`, it is a good idea to double-check the result of the **Vertices** command before using this procedure.

```
> Vertices(firstOrdered);
                    ["a", "b", "c", "d", "e", "f", "g"]                    (11.23)
> SetChildOrder(firstOrdered,[0,2,1,3,2,1,1]);
```

A type for ordered rooted trees

Now that we've ordered the children in Maple's representation of this tree, `firstOrdered` now represents an ordered rooted tree. We will create an `ORTree` type. The requirement for being an ordered rooted tree are: the object must be a rooted tree, every vertex must have an "order" attribute set, and the root's "order" must be 0.

```
> `type/ORTree` := proc(obj)
    local v;
    uses GraphTheory;
    if not type(obj,RTree) then
      return false;
    end if;
    for v in Vertices(obj) do
      if GetVertexAttribute(obj,v,"order") = FAIL then
        return false;
      end if;
    end do;
    v := GetGraphAttribute(obj,"root");
    if GetVertexAttribute(obj,v,"order") <> 0 then
      return false;
    end if;
    return true;
  end proc:
> type(firstOrdered,ORTree);
```

true **(11.24)**

Drawing ordered rooted trees

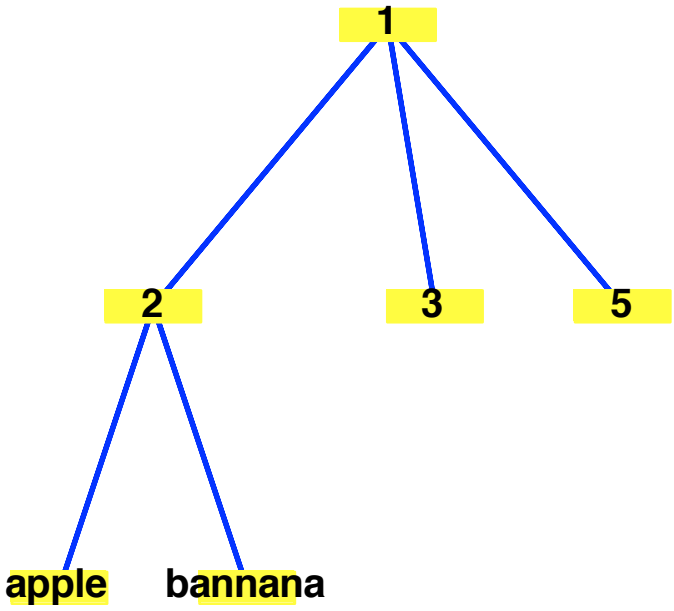
While **firstOrdered** is now officially an ordered rooted tree and is storing the order of children, if we draw it, it will not appear with children in the proper order. To accomplish this, we need to create a **DrawORTree** procedure. Recall that when creating the **DrawRTree** procedure, our basic approach was to have Maple determine the correct positions of vertices for us by turning the rooted tree into a graph that it would draw in the manner we wished. We do the same thing in this case.

The key fact that we'll use is that, in fact, Maple draws trees with the vertices in a fixed order. For example,

```
> orderEx1 := Graph({{1,2},{1,5},{1,3},{2,"bannana"},{2,
  "apple"}});
```

orderEx1 := Graph 5: an undirected unweighted graph with 6 vertices and 5 edge(s) **(11.25)**

```
> DrawGraph(orderEx1);
```



Despite the order in which we listed the edges, Maple sorted the vertices with numeric names in numeric order and it sorted the vertices with string names in lexicographic order. This is because we provided only the edges and made Maple determine the graph's vertices for itself. Notice that when we use the **Vertices** command, the vertices are listed in the same order as they appear in the graph.

```
> Vertices(orderEx1);
```

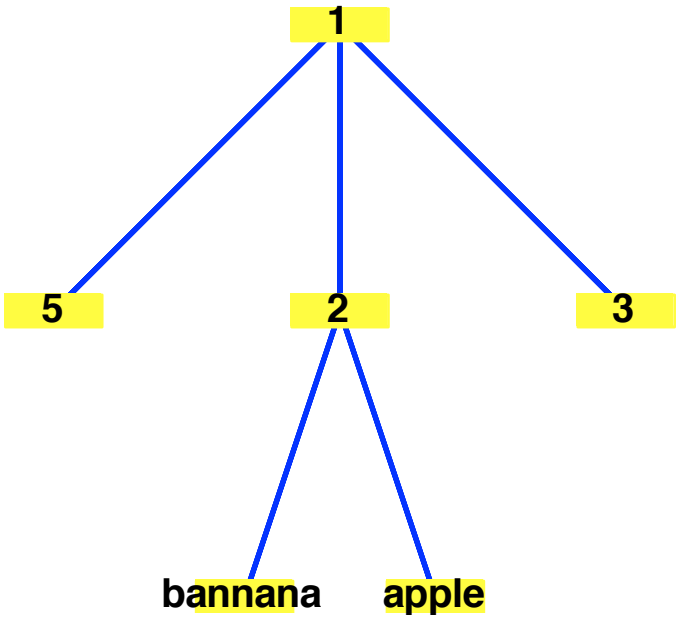
[1, 2, 3, 5, "apple", "bannana"] **(11.26)**

But if we give the **Graph** command the list of vertices explicitly and in the order we want them, we can change the graph.

```
> orderEx2 := Graph([1,5,2,3,"bannana","apple"],{{1,2},{1,5},
  {1,3},{2,"bannana"},{2,"apple"}});
```

orderEx2 := Graph 6: an undirected unweighted graph with 6 vertices and 5 edge(s) **(11.27)**

```
> DrawGraph(orderEx2);
```



We will use this feature in order to properly draw ordered rooted graphs. Given an **ORTree**, we'll use the **UnderlyingGraph** to access the undirected version of the graph, in particular, the undirected edges. We then determine an order for the vertices that is compatible with the ordering of children. And then we create a new graph using the order of the vertices and the edges from the underlying graph. Finally, as with **DrawRTree**, we use the vertex positions of that tree to draw our **ORTree**.

The key is to create the ordered list of vertices, which we'll call **OVerts**. We will use an approach similar to how we determined that all edges were in the correct direction in the **RTree** type definition. We initialize **OVerts** to the list containing only the root and we initialize a counter **i** to 1. We then use the **Departures** command to find all of the children of the root. (We could also use the **FindChildren** command we defined earlier, but using **Departures** directly is a bit more efficient.) We then sort the children in order of their "order" attribute. Once sorted, we add the children 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 according to their "order" attribute and added to the list.

Before writing the main procedure, we'll create a helper procedure to aid in the sorting of children. Recall that passing the **sort** command a procedure as its optional second argument allows us to specify the order in which it arranges values. This procedure must be boolean-valued and should return true if the first argument precedes the second argument. In our case, we need a procedure that compares two vertices by comparing their "order" attribute. But note that we can't access the vertex attribute without also having the name of the graph, since **GetVertexAttribute** requires the name of the graph. In other words, the procedure for comparing vertices depends on the graph, but we are only allowed to have two arguments in the procedure. So we'll create a function that takes a graph and returns a comparison procedure for that graph.

```
> VOrderComp := G -> proc(u,v)
    local uOrder, vOrder;
    uOrder := GraphTheory[GetVertexAttribute](G,u,"order");
    vOrder := GraphTheory[GetVertexAttribute](G,v,"order");
```

```
    if uOrder < vOrder then
      return true;
    else
      return false;
    end if;
  end proc:
```

Observe that `VOrderComp` applied to our `firstOrdered` example returns a procedure that we can use as the optional argument to **sort**.

```
> sort(["b","c","d"],VOrderComp(firstOrdered));
```

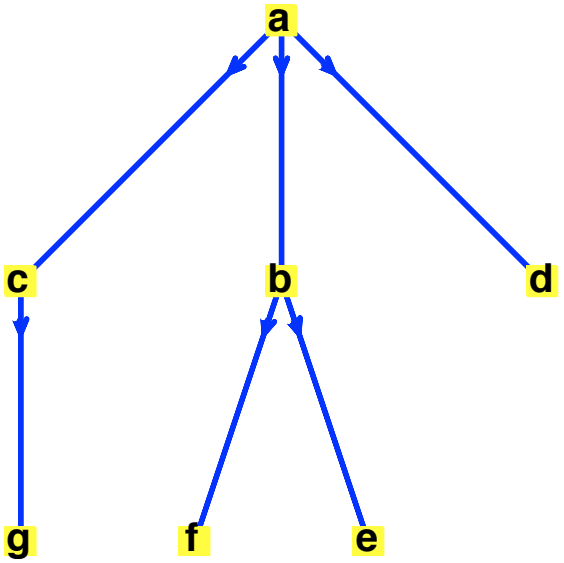
["c", "b", "d"] **(11.28)**

Here, now, is the procedure `DrawORTree`.

```
> DrawORTree := proc(T::ORTree)
    local E, sorter, R, OVerts, i, numverts, children, G, VP,
  v, p;
    uses GraphTheory;
    E := Edges(UnderlyingGraph(T));
    sorter := VOrderComp(T);
    R := GetGraphAttribute(T,"root");
    OVerts := [R];
    i := 1;
    while i <= nops(OVerts) do
      children := Departures(T,OVerts[i]);
      children := sort(children,sorter);
      OVerts := [op(OVerts),op(children)];
      i := i + 1;
    end do;
    G := Graph(OVerts,E);
    DrawGraph(G,style=tree,root=R);
    VP := GetVertexPositions(G);
    for i from 1 to nops(Vertices(T)) do
      v := OVerts[i];
      p := VP[i];
      SetVertexAttribute(T,v,"draw-pos-user"=p);
    end do;
    DrawGraph(T);
  end proc:
```

Finally, we can draw our ordered tree correctly.

```
> DrawORTree(firstOrdered);
```



Properties of Trees

We conclude this section with procedures 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 a 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.

```
> FindLevel := proc(T::RTree, targetV)
    local v, level, root;
    uses GraphTheory;
    level := 0;
    v := targetV;
    root := GetGraphAttribute(T,"root");
    while v <> root do
      level := level + 1;
      v := FindParent(T,v);
    end do;
    return level;
  end proc:
```

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

```
> FindLevel(firstOrdered,"g");
```

2 **(11.29)**

```
> FindLevel(firstOrdered,"d");
```

1 **(11.30)**

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.

```
> FindHeight := proc(T::RTree)
    local height, v, level;
```

```
    uses GraphTheory;
    height := 0;
    for v in Vertices(T) do
      level := FindLevel(T,v);
      if level > height then
        height := level;
      end if;
    end do;
    return height;
  end proc:
> FindHeight(firstOrdered);
```

$$2 \qquad \textbf{(11.31)}$$

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** procedure we wrote earlier, (3) test each leaf's level and return false if it is higher than level $h-1$.

```
> IsBalanced := proc(T::RTree)
    local height, leaves, v;
    uses GraphTheory;
    height := FindHeight(T);
    leaves := FindLeaves(T);
    for v in leaves do
      if FindLevel(T,v) < height - 1 then
        return false;
      end if;
    end do;
    return true;
  end proc:
```

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

```
> IsBalanced(firstOrdered);
```

$$\mathit{true} \qquad \textbf{(11.32)}$$

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 Maple

Before we get to the applications, we will discuss how we can represent binary trees in Maple and develop some procedures 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 type

We will construct a type, **BTree**, for binary trees. We will impose three conditions for an object to be considered a **BTree**. First, it must be an ordered rooted tree, *i.e.*, a **ORTree**. 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.

```
> firstBTree := Graph(["D","B","F","A","C","E"],{["D","B"],
  ["D","F"],["B","A"],["B","C"],["F","E"]});
```

firstBTree := Graph 7: a directed unweighted graph with 6 vertices and 5 arc(s) **(11.33)**

```
> SetGraphAttribute(firstBTree,"root"="D");
> SetVertexAttribute(firstBTree,"D","order"=0);
> SetVertexAttribute(firstBTree,"B","order"=1);
> SetVertexAttribute(firstBTree,"F","order"=2);
> SetVertexAttribute(firstBTree,"A","order"=1);
> SetVertexAttribute(firstBTree,"C","order"=2);
> SetVertexAttribute(firstBTree,"E","order"=1);
```

Now that we have an example, let's create the type. To check that the tree is in fact binary, we can use the **Departures** command and count the number of children of each vertex. If any vertex has more than two children, then it 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.

```
> `type/BTree` := proc(obj)
    local R, v, vpos;
    uses GraphTheory;
    if not type(obj,ORTree) then
      return false;
    end if;
    R := GetGraphAttribute(obj,"root");
    for v in Vertices(obj) do
      if nops(Departures(obj,v)) > 2 then
        return false;
      end if;
      vpos := GetVertexAttribute(obj,v,"order");
      if (vpos = 0 and v <> R) or
         ((vpos <> 0) and (vpos <> 1 ) and (vpos <> 2)) then
       return false;
      end if;
    end do;
    return true;
  end proc:
> type(firstBTree,BTree);
```

true **(11.34)**

Drawing binary trees

Next we'll write a procedure for drawing binary trees. Note that, while **DrawORTree** will draw a binary tree, that procedure will display vertices that are "only children" directly below their parent rather than to the left or right. We will explicitly calculate 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-

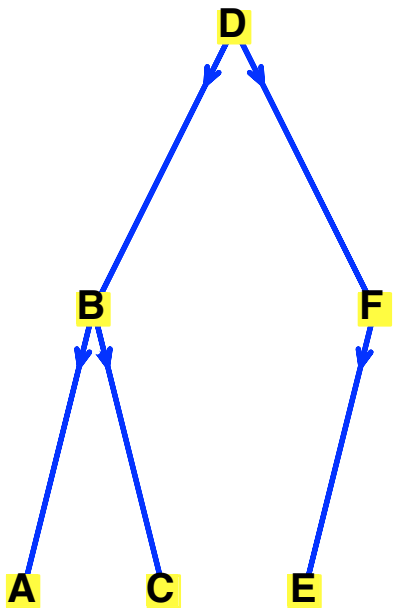
coordinate of each vertex will depend on the height of the tree and the level of the vertex. Specifically, the y coordinate of any vertex is $1 - l/h$, where l is the level of the vertex and h is the height of the tree. This way, the root, which is at level 0, has y-coordinate 1 and the vertices in the 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 $1/2$. The left child of the root will be drawn at x-coordinate $1/4$ and the right child at $3/4$. We can think about the children of the root as being drawn $1/4$ to the left of the root and $1/4$ 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 left children) or minus (for right) $\frac{1}{2^{l+1}}$.

The **DrawBTree** procedure is below. It begins by calculating the height of the tree with the **FindHeight** procedure we created earlier. It then processes the root of the tree by setting its position to $(1/2, 1)$. (Note that we set the position of the vertex by setting the "draw-pos-user" attribute of the vertex. This is in some ways more convenient than assembling all of the vertex positions and then using **SetVertexPositions**.) We also create a list, **Verts**, and populate it with the root's children. We then begin a loop with the same structure as the while loop in the **RTree** type definition. We have a counter **i** initialized to 1. This counter serves as an index into the **Verts** list. At each step in the while loop, we do several things. First, we use the **FindLevel** procedure we created earlier to determine the level of the vertex. Second, the y-coordinate is calculated by the formula $1 - l/h$. 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 "draw-pos-user" attribute for the vertex. And finally, the children of the current vertex are added to the **Verts** list and the counter is incremented.

```
> DrawBTree := proc(T::BTree)
    local height, v, i, level, Verts, x, y, parent, side;
    uses GraphTheory;
    height := FindHeight(T);
    v := GetGraphAttribute(T,"root");
    SetVertexAttribute(T,v,"draw-pos-user"=[1/2,1]);
    Verts := FindChildren(T,v);
    i := 1;
    while i <= nops(Verts) do
      v := Verts[i];
      level := FindLevel(T,v);
      y := 1 - level/height;
      parent := FindParent(T,v);
      x := GetVertexAttribute(T,parent,"draw-pos-user")[1];
      if GetVertexAttribute(T,v,"order") = 1 then
        side := -1;
      else
        side := 1;
      end if;
      x := x + side * 1/(2^(level+1));
      SetVertexAttribute(T,v,"draw-pos-user"=[x,y]);
      Verts := [op(Verts),op(FindChildren(T,v))];
```

```
      i := i + 1;
    end do;
    DrawGraph(T);
  end proc:
> DrawBTree(firstBTree);
```



Parents and children

In the previous section, we created the procedure **FindParent**, which returns the parent of a given vertex in the given tree. This procedure works on **BTrees** as well.

```
> FindParent(firstBTree,"C");
```

"B" **(11.35)**

We had also created the **FindChildren** procedure, which we could 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 procedure (respectively, 2 and right child). If there is no left (respectively, right), child, the procedure will return **FAIL**.

```
> FindLeftChild := proc(T::BTree, v)
    local children, w, pos;
    uses GraphTheory;
    children := FindChildren(T,v);
    for w in children do
      pos := GetVertexAttribute(T,w,"order");
      if pos = 1 then
        return w;
      end if;
    end do;
    return FAIL;
  end proc:
> FindRightChild := proc(T::BTree, v)
    local children, w, pos;
    uses GraphTheory;
    children := FindChildren(T,v);
```

```
    for w in children do
      pos := GetVertexAttribute(T,w,"order");
      if pos = 2 then
        return w;
      end if;
    end do;
    return FAIL;
  end proc:
> FindLeftChild(firstBTree,"F");
                                "E"                    (11.36)
> FindRightChild(firstBTree,"F");
                                FAIL                   (11.37)
```

Building binary trees

We will also want procedures to create and build up a binary tree. Specifically, we'll create a procedure that, given the label for the root of a binary tree, creates the tree with that vertex as its root. We will then write procedures 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 **NewBTree** procedure creates the binary tree consisting of a single vertex, the root of the tree.

```
> NewBTree := proc(R)
    local T;
    uses GraphTheory;
    T := Digraph([R]);
    SetGraphAttribute(T,"root"=R);
    SetVertexAttribute(T,R,"order"=0);
    return T;
  end proc:
```

Adding a child to a vertex in a binary tree requires three basic steps. We must add a vertex to the graph with the **AddVertex** command, add a directed edge from the parent to the new vertex with the **AddArc** command, and then use **SetVertexAttribute** to identify the new child as left or right by setting the "order" attribute to 1 or 2, respectively.

In Maple 14, however, we must do some additional work. Recall that the **AddVertex** command does not modify the original graph, but instead creates a new graph whose vertices are the original vertices together with the new vertex and whose edge set is identical to the original. The **AddVertex** command, however, does not preserve attributes. As a result, we will need to manually copy the "root" attribute for the graph and the "order" attributes on each vertex.

```
> AddLeftChild := proc(T::BTree, v, newV)
    local newT, newedge, temp, w;
    uses GraphTheory;
    if FindLeftChild(T,v) <> FAIL then
      error "Vertex already has a left child.";
    end if;
    newT := AddVertex(T,newV);
    AddArc(newT,[v,newV]);
    temp := GetGraphAttribute(T,"root");
    SetGraphAttribute(newT,"root"=temp);
    for w in Vertices(T) do
      temp := GetVertexAttribute(T,w,"order");
      SetVertexAttribute(newT,w,"order"=temp);
    end do;
```

```
    SetVertexAttribute(newT,newV,"order"=1);
    return newT;
  end proc:
> AddRightChild := proc(T::BTree, v, newV)
    local newT, newedge, temp, w;
    uses GraphTheory;
    if FindRightChild(T,v) <> FAIL then
      error "Vertex already has a left child.";
    end if;
    newT := AddVertex(T,newV);
    AddArc(newT,[v,newV]);
    temp := GetGraphAttribute(T,"root");
    SetGraphAttribute(newT,"root"=temp);
    for w in Vertices(T) do
      temp := GetVertexAttribute(T,w,"order");
      SetVertexAttribute(newT,w,"order"=temp);
    end do;
    SetVertexAttribute(newT,newV,"order"=2);
    return newT;
  end proc:
```

With **NewBTree**, **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.

```
> Fig1Tree := NewBTree("Math");
```

Fig1Tree := Graph 8: a graph with 1 vertex and no edges **(11.38)**

```
> Fig1Tree := AddRightChild(Fig1Tree,"Math","Phys");
```

Fig1Tree := Graph 9: a directed unweighted graph with 2 vertices and 1 arc(s) **(11.39)**

```
> Fig1Tree := AddLeftChild(Fig1Tree,"Math","Geog");
```

Fig1Tree := Graph 10: a directed unweighted graph with 3 vertices and 2 arc(s) **(11.40)**

```
> Fig1Tree := AddRightChild(Fig1Tree,"Phys","Zoo");
```

Fig1Tree := Graph 11: a directed unweighted graph with 4 vertices and 3 arc(s) **(11.41)**

```
> Fig1Tree := AddLeftChild(Fig1Tree,"Phys","Meteo");
```

Fig1Tree := Graph 12: a directed unweighted graph with 5 vertices and 4 arc(s) **(11.42)**

```
> Fig1Tree := AddRightChild(Fig1Tree,"Geog","Geol");
```

Fig1Tree := Graph 13: a directed unweighted graph with 6 vertices and 5 arc(s) **(11.43)**

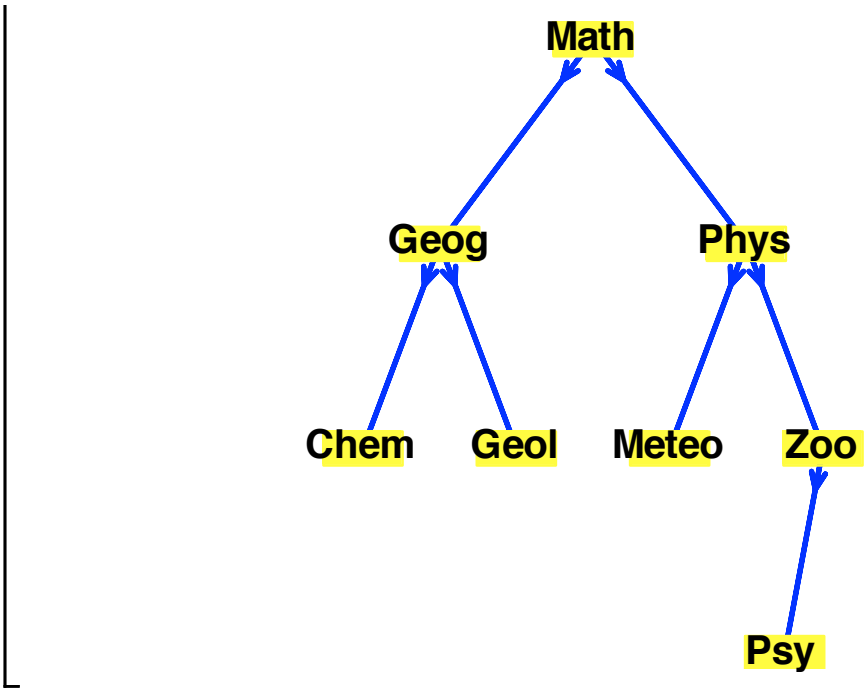
```
> Fig1Tree := AddLeftChild(Fig1Tree,"Zoo","Psy");
```

Fig1Tree := Graph 14: a directed unweighted graph with 7 vertices and 6 arc(s) **(11.44)**

```
> Fig1Tree := AddLeftChild(Fig1Tree,"Geog","Chem");
```

Fig1Tree := Graph 15: a directed unweighted graph with 8 vertices and 7 arc(s) **(11.45)**

```
> DrawBTree(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 procedure `BInsertion`.

The `BInsertion` procedure will accept two input values: a binary search tree and a vertex to be found or added. The procedure 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 by checking the tree's "root" attribute and setting the local variable `v` to the root. Then we begin a while loop. This while loop continues provided two conditions are met. First, that `v <> 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 <> 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 Maple identifies a vertex with its label, so, unlike the text, we do not distinguish between v and $label(v)$.)

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.

The evaln parameter modifier

Before providing the definition of the insertion procedure, we also mention the **evaln** parameter modifier. We want the **BInsertion** procedure to be able to modify the tree it was given. Normally when you provide a parameter to a procedure, the procedure cannot assign to the parameter. Consider the following simple procedure.

```
> five := 5;
```

$$five := 5 \qquad (11.46)$$

```
> AddOne := proc(n)
    n := n + 1;
  end proc:
> AddOne(five);
Error, (in AddOne) illegal use of a formal parameter
```

This procedure produces an error when we attempt to assign a value to the parameter **n**. This is a feature of programming in Maple that is designed to encourage good programming practices. In particular, for a procedure 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 a parameter can cause serious bugs in your programs.

You can think about what's going on in the **AddOne** procedure this way: when you call the procedure 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 **evaln** modifier provides a way around this. For example, if we declare the parameter to be **n::evaln** in the **AddOne** procedure, we are telling Maple to not evaluate the parameter **n** into the object that it refers to, but to evaluate the parameter into a name. Instead of evaluating the variable **five** to get the object **5** and replacing the parameter **n** with **5**, the variable **five** is evaluated into the name **five** and the parameter **n** is replaced with the name **five**. This means that the command becomes **five := five + 1;**. This still doesn't quite work, because the **five** on the right hand side of the assignment is now a name, and we can't add integers to names. We need to force the name **five** to be evaluated to the object 5 with the **eval** command.

```
> AddOne2 := proc(n::evaln)
    n := eval(n) + 1;
  end proc:
```

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

```
> seven := 7;
```

$$seven := 7 \qquad (11.47)$$

```
> AddOne2(seven);
```

$$8 \qquad (11.48)$$

```
> seven;
```

$$8 \qquad (11.49)$$

To summarize: we can create procedures that modify a variable that is passed to them by marking the parameter with the modifier **evaln**. This allows the parameter to appear on the right side of an assignment, but the trade-off is that all other occurrences of the parameter must have an explicit **eval**.

Implementation of binary insertion

Here, now, is the binary insertion algorithm.

```
> BInsertion := proc(BST::evaln,x)
    local v;
    uses GraphTheory;
    if not type(eval(BST),BTree) then
      BST := NewBTree(x);
      return false;
    end if;
    v := GetGraphAttribute(eval(BST),"root");
    while v <> NULL and v <> x do
      if x < v then
        if FindLeftChild(eval(BST),v) = FAIL then
          BST := AddLeftChild(eval(BST),v,x);
          v := NULL;
        else
          v := FindLeftChild(eval(BST),v);
        end if;
      else
        if FindRightChild(eval(BST),v) = FAIL then
          BST := AddRightChild(eval(BST),v,x);
          v := NULL;
        else
          v := FindRightChild(eval(BST),v);
        end if;
      end if;
    end do;
    if v = NULL then
      return false;
    else
      return true;
    end if;
  end proc:
```

Note that we begin the procedure by testing to see if the object stored in the parameter **BST** is in fact a binary tree. If it is not, we use the **NewBTree** command to create a tree and store it in the name given by the parameter. This way we can use **BInsertion** to create a new tree in addition to inserting elements in an existing tree.

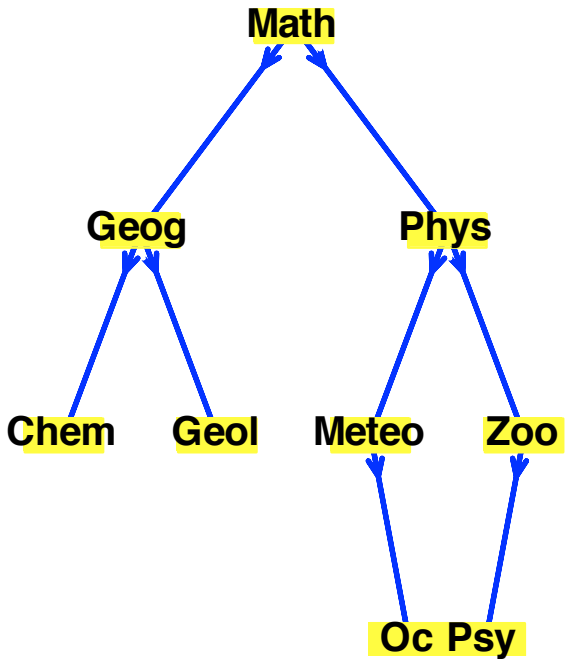
Now, let's see if oceanography is in the **Fig1Tree** of academic subjects.

```
> BInsertion(Fig1Tree,"Oc");
```

false **(11.50)**

The procedure returned false indicating that "Oc" was not found in the tree. Graphing **Fig1Tree**, we see that it was added as a child of meteorology.

```
> DrawBTree(Fig1Tree);
```

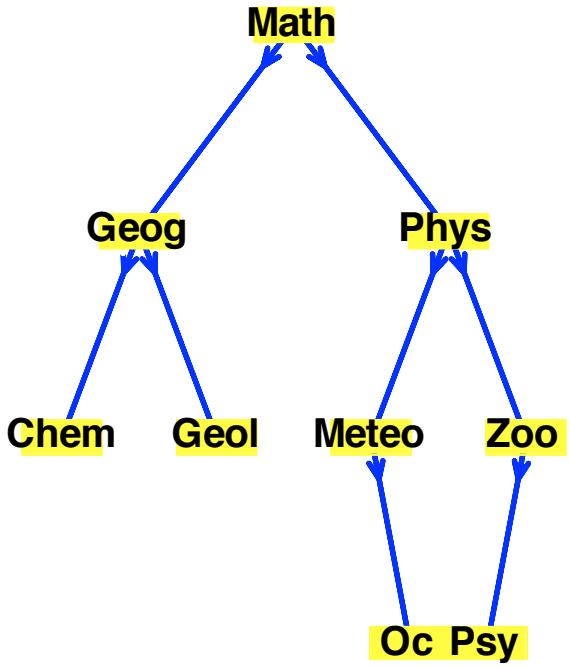


On the other hand, zoology is already in the tree and so the tree is not modified.

```
> BInsertion(Fig1Tree,"Zoo");
```

true **(11.51)**

```
> DrawBTree(Fig1Tree);
```



Constructing a binary search tree from a list

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

```
> MakeBST := proc(L::list)
    local T, v;
    for v in L do
```

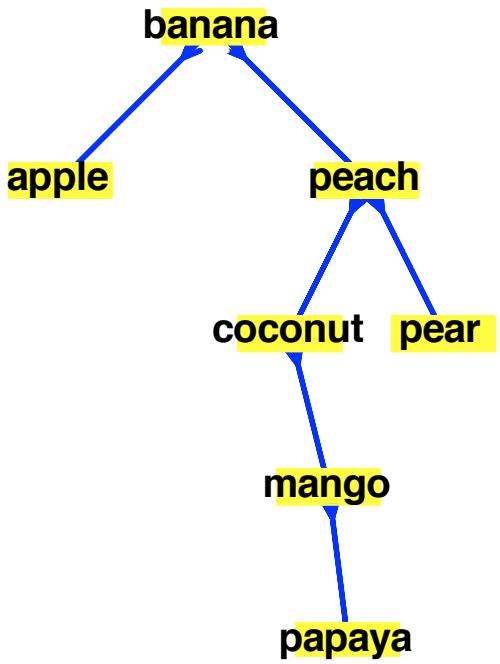
```
      BInsertion(T,v);
    end do;
    return T;
  end proc:
```

We use this to complete Exercise 1 from Section 11.2.

```
> Exercise1 := MakeBST(["banana", "peach", "apple", "pear",
  "coconut", "mango", "papaya"]);
```

Exercise1 := Graph 16: a directed unweighted graph with 7 vertices and 6 arc(s) **(11.52)**

```
> DrawBTree(Exercise1);
```



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 will create a procedure, similar to `NewBTree`, which, in addition to creating the binary tree, also assigns a "weight" attribute to the graph to store the weight of the symbol.

```
> NewHTree := proc(s,w)
    local T;
    uses GraphTheory;
    T := Digraph([s]);
    SetGraphAttribute(T,"root"=s);
    SetGraphAttribute(T,"weight"=w);
    SetVertexAttribute(T,s,"order"=0);
    return T;
  end proc:
```

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

```
> CreateForest := proc(L::list(list))
    local forest, M, v, w, G;
    uses GraphTheory;
    forest := [];
    for M in L do
      v := M[1];
      w := M[2];
      forest := [op(forest),NewHTree(v,w)];
    end do;
    return forest;
  end proc:
```

Using this procedure, we form the initial forest for Exercise 23 from Section 11.2.

```
> Ex23Forest := CreateForest([["a",0.20],["b",0.10],["c",0.15],
  ["d",0.25],["e",0.30]]);
```

Ex23Forest := [*Graph 17: a graph with 1 vertex and no edges*, **(11.53)**

Graph 18: a graph with 1 vertex and no edges,

Graph 19: a graph with 1 vertex and no edges,

Graph 20: a graph with 1 vertex and no edges,

Graph 21: a graph with 1 vertex and no edges]

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 lowest weight and second lowest weight trees 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** command accepts an optional second argument, specifically, a procedure on two arguments that returns true if the first argument precedes the second in the desired order. The following procedure will be of this kind, accepting two binary trees as input and comparing their weights.

```
> CompareTrees := proc(A::BTree,B::BTree)
    local a, b;
    uses GraphTheory;
    a := GetGraphAttribute(A,"weight");
    b := GetGraphAttribute(B,"weight");
    return evalb(a < b);
  end proc:
> Ex23Forest := sort(Ex23Forest,CompareTrees);
```

Ex23Forest := [*Graph 18: a graph with 1 vertex and no edges*, **(11.54)**

Graph 19: a graph with 1 vertex and no edges,

Graph 17: a graph with 1 vertex and no edges,

Graph 20: a graph with 1 vertex and no edges,

Graph 21: a graph with 1 vertex and no edges]

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 procedure 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 tress 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) setting the graph attribute "root" and setting the "weight" attribute to be the sum of the weights of the two original trees. Also, (5) we will use edge weights of 0 and 1 to label the edges, so the edge weights also need to be copied for the original edges and added to the new edges.

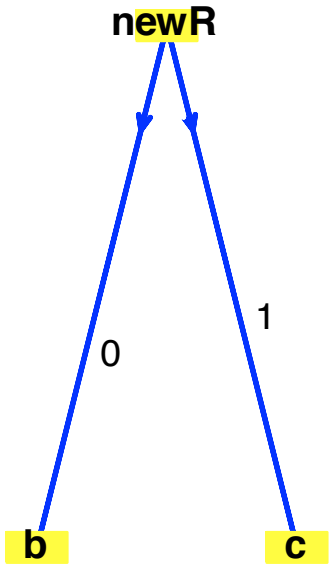
```
> JoinHTrees := proc(newR,A::BTree,B::BTree)
    local newT, newVerts, Aroot, Broot, newEdges, v, e, p, w;
    uses GraphTheory;
    newVerts := [newR,op(Vertices(A)),op(Vertices(B))];
    Aroot := GetGraphAttribute(A,"root");
    Broot := GetGraphAttribute(B,"root");
    newEdges := Edges(A) union Edges(B)
                         union {[newR,Aroot],[newR,Broot]};
    newT := Graph(weighted,newVerts,newEdges);
    for v in Vertices(A) do
      p := GetVertexAttribute(A,v,"order");
      SetVertexAttribute(newT,v,"order"=p);
    end do;
    for v in Vertices(B) do
      p := GetVertexAttribute(B,v,"order");
      SetVertexAttribute(newT,v,"order"=p);
    end do;
    for e in Edges(A) do
      p := GetEdgeWeight(A,e);
      SetEdgeWeight(newT,e,p);
    end do;
    for e in Edges(B) do
      p := GetEdgeWeight(B,e);
      SetEdgeWeight(newT,e,p);
    end do;
    SetVertexAttribute(newT,Aroot,"order"=1);
    SetVertexAttribute(newT,Broot,"order"=2);
    SetVertexAttribute(newT,newR,"order"=0);
    SetEdgeWeight(newT,[newR,Aroot],0);
    SetEdgeWeight(newT,[newR,Broot],1);
    SetGraphAttribute(newT,"root"=newR);
    w := GetGraphAttribute(A,"weight")
                              + GetGraphAttribute(B,"weight");
    SetGraphAttribute(newT,"weight"=w);
    return newT;
  end proc:
```

For example, we'll join the first two graphs in our sorted `Ex23Forest`.

```
> exampleJoin:=JoinHTrees("newR",Ex23Forest[1],Ex23Forest[2]);
```

exampleJoin := Graph 22: a directed weighted graph with 3 vertices and 2 arc(s) **(11.55)**

```
> DrawBTree(exampleJoin);
```



Implementing the main procedure

We now have the major pieces of the Huffman algorithm assembled and we can write the `HuffmanCode` algorithm. This algorithm will accept as input the same list of symbol-weight pairs as the `CreateForest` procedure did. The procedure'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 member. Inside the while loop, we first use the `CompareTrees` procedure to sort the forest in increasing order of weight. Then, we use the `JoinTrees` procedure to join the first two trees in the forest and we add that new tree to the list, replacing the original two.

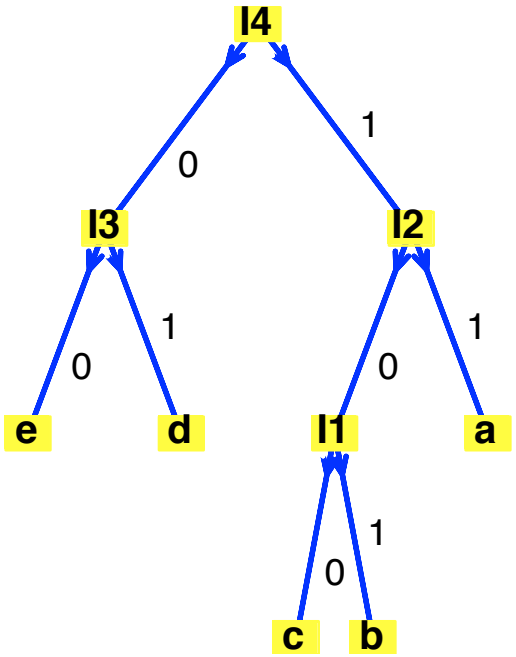
```
> HuffmanCode := proc(L::list(list))
    local F, i, tempT;
    uses GraphTheory;
    F := CreateForest(L);
    i := 0;
    while nops(F) > 1 do
      F := sort(F,CompareTrees);
      i := i + 1;
      tempT := JoinHTrees(cat("I",i), F[2],F[1]);
      F := [op(F[3..-1]),tempT];
    end do;
    return F[1];
  end proc:
```

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 `cat` to create the names of the internal vertices.

```
> Ex23HCode := HuffmanCode([["a",0.20],["b",0.10],["c",0.15],
  ["d",0.25],["e",0.30]]);
```

Ex23HCode := Graph 23: a directed weighted graph with 9 vertices and 8 arc(s) **(11.56)**

```
> DrawBTree(Ex23HCode);
```



Encoding strings using the Huffman code tree

We conclude this section by writing a procedure to encode a string of symbols using a given Huffman tree. Note that you can use the list accessor notation on a string to access its individual characters. For example,

```
> exampleString := "Hello";
```

exampleString := "Hello" **(11.57)**

```
> exampleString[2];
```

"e" **(11.58)**

Since we encode a string by assembling the codes for individual characters, we'll start with a procedure 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.

```
> EncodeCharacter := proc(H::BTree,c::character)
    local code, vertex, parent, digit;
    uses GraphTheory;
    vertex := c;
    code := "";
    while FindParent(H,vertex) <> FAIL do
      parent := FindParent(H,vertex);
      digit := GetEdgeWeight(H,[parent,vertex]);
      code := cat("",digit,code);
      vertex := parent;
    end do;
    return code;
  end proc:
```

```
> EncodeCharacter(Ex23HCode,"c");
```

"100" **(11.59)**

(Note: in updating **code**, we begin the concatenation with the empty string, **""**. This is because

cat returns an object of the same type as its first argument. If we did not begin with the empty string, the result would not be a string.

To encode a string, we encode each character and assemble the results. We also make use of the **length** command in this procedure. Applied to a string, **length** returns the number of characters in the string.

```
> EncodeString := proc(H::BTree,S::string)
    local code, charcode, i;
    code := "";
    for i from 1 to length(S) do
      charcode := EncodeCharacter(H,S[i]);
      code := cat(code,charcode);
    end do;
    return code;
  end proc:
```

We use this to encode the word "ace".

```
> EncodeString(Ex23HCode,"ace");
```

"1110000" **(11.60)**

11.3 Tree Traversal

In this section we show how to use Maple 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. An **ORTree** is an **RTree** with the additional restriction that each vertex has an "order" attribute.

Also recall that we created the function **VOrderComp**, which takes an ordered rooted tree and returns a procedure that compares vertices based on their "order" attribute. The procedure returned by **VOrderComp** can be used as an optional argument to **sort** to sort lists of vertices based on the "order" attribute.

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.

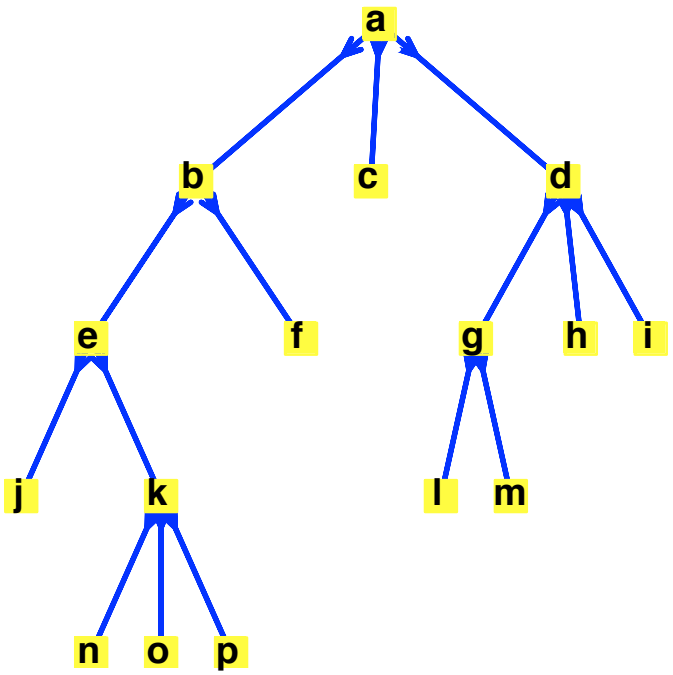
```
> Fig3ORTree := Graph({["a","b"],["a","c"],["a","d"],["b","e"],
  ["b","f"],["d","g"],["d","h"],["d","i"],["e","j"],["e","k"],
  ["g","l"],["g","m"],["k","n"],["k","o"],["k","p"]});
```

Fig3ORTree := Graph 24: a directed unweighted graph with 16 vertices and 15 arc(s) **(11.61)**

```
> SetGraphAttribute(Fig3ORTree,"root"="a");
> Vertices(Fig3ORTree);
```

["a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p"] **(11.62)**

```
> SetChildOrder(Fig3ORTree,[0,1,2,3,1,2,1,2,3,1,2,1,2,1,2,3]);
> DrawORTree(Fig3ORTree);
```



Subtrees

Before implementing the traversal algorithms, we need a procedure that determines a subtree of a tree. In particular, we want an algorithm 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 **InducedSubgraph** command on the list of vertices in the desired subtree. (Note that, unlike the **AddVertex** command, **InducedSubgraph** preserves the vertex attributes of the vertices in the subgraph.) The vertices that we want included in the subgraph are the given vertex together with all of its descendents. We begin by creating a procedure that finds all of the descendants of the given vertex. This procedure 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 **Departures** command. 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.)

```
> Descendants := proc(T::RTree, parent)
    local Dlist, v, i;
    uses GraphTheory;
    Dlist := Departures(T,parent);
    i := 1;
    while i <= nops(Dlist) do
      v := Dlist[i];
      Dlist := [op(Dlist),op(Departures(T,v))];
      i := i + 1;
    end do;
    return Dlist;
  end proc:
```

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

```
> Descendants(Fig3ORTree,"e");
```

["j", "k", "n", "o", "p"] **(11.63)**

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 **InducedSubgraph** command on the given vertex and all its descendants; set the "root" attribute of the subgraph; and set the "order" attribute of the root to 0.

```
> SubTree := proc(T::ORTree, newRoot)
    local Vlist, subT;
    uses GraphTheory;
    Vlist := Descendants(T,newRoot);
    Vlist := [newRoot,op(Vlist)];
    subT := InducedSubgraph(T,Vlist);
    SetGraphAttribute(subT,"root"=newRoot);
    SetVertexAttribute(subT,newRoot,"order"=0);
    return subT;
  end proc:
```

Let's check this procedure by finding the subtree with root *e* and make sure it really is an **ORTree**.

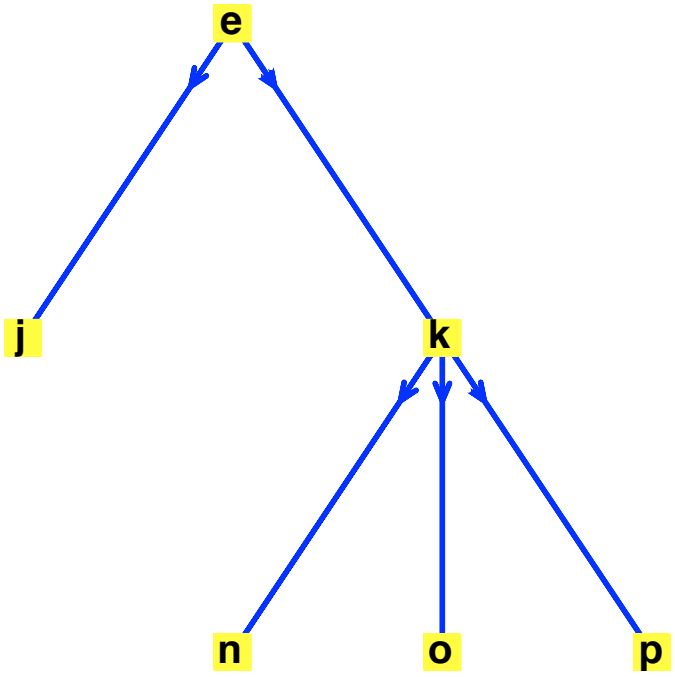
```
> subEx := SubTree(Fig3ORTree,"e");
```

subEx := Graph 25: a directed unweighted graph with 6 vertices and 5 arc(s) **(11.64)**

```
> type(subEx,ORTree);
```

true **(11.65)**

```
> DrawORTree(subEx);
```



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

```
> Preorder := proc(T::ORTree)
    local root, children, sorter, i, tempSubT;
    uses GraphTheory;
    root := GetGraphAttribute(T,"root");
    printf("%a ",root);
    children := Departures(T,root);
    sorter := VOrderComp(T);
    children := sort(children,sorter);
    for i from 1 to nops(children) do
      tempSubT := SubTree(T,children[i]);
      Preorder(tempSubT);
    end do;
  end proc:
> Preorder(Fig3ORTree);
"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.

```
> Postorder := proc(T::ORTree)
    local root, children, sorter, i, tempSubT;
    uses GraphTheory;
    root := GetGraphAttribute(T,"root");
    children := Departures(T,root);
    sorter := VOrderComp(T);
    children := sort(children,sorter);
    for i from 1 to nops(children) do
      tempSubT := SubTree(T,children[i]);
      Postorder(tempSubT);
    end do;
    printf("%a ",root);
  end proc:
> Postorder(Fig3ORTree);
"j" "n" "o" "p" "k" "e" "f" "b" "c" "l" "m" "g" "h" "i" "d"
"a"
```

Inorder

In inorder traversal, the algorithm first applies 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.

```
> Inorder := proc(T::ORTree)
    local root, children, sorter, i, tempSubT;
    uses GraphTheory;
    root := GetGraphAttribute(T,"root");
    children := Departures(T,root);
    sorter := VOrderComp(T);
    children := sort(children,sorter);
    if nops(children) <> 0 then
      tempSubT := SubTree(T,children[1]);
      Inorder(tempSubT);
```

```
    end if;
    printf("%a ",root);
    for i from 2 to nops(children) do
      tempSubT := SubTree(T,children[i]);
      Inorder(tempSubT);
    end do;
  end proc:
> Inorder(Fig3ORTree);
"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 Maple 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 Maple procedure 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. In particular, we must address a few issues.

First, we need to represent arithmetic expressions in Maple in such a way that we can work with them and ensure that Maple won't evaluate them.

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 Maple to consider each of these to be a distinct object, since Maple insists that the vertices in a graph be distinct.

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 procedure to perform the combination of subtrees described in part (2) of the recursive step.

Representing expressions

We cannot use the usual operations of arithmetic to represent an expression that will be turned into a tree, as Maple will automatically perform arithmetic operations. In order to prevent evaluation, we will use different operators, namely, `&+`, `&-`, `&*`, `&/`, and `&^`.

Recall that the ampersand indicates to Maple that the symbol is a neutral operator. Ordinarily, you

use neutral operators to define new infix operators. In this case, we only need to be able to write expressions in terms of infix operators and have the expression not evaluated. As long as we don't define `&+`, `&-`, `&*`, `&/`, or `&^`, Maple will understand that they are operators but will not evaluate them.

Note that the typical order of operations, or precedence, is not respected by these neutral operators. So to enter expressions with these operators, we must fully parenthesize the expression. For example, to enter $2 \cdot 3 + 4$, type the following.

```
> (2 &* 3) &+ 4;
```

$$(2 \,\&*\, 3) \,\&+\, 4 \qquad \textbf{(11.66)}$$

We will impose some additional restrictions on expressions, in order to avoid unnecessary complications. Specifically, we insist that the only symbols allowed are integers, variables, the binary neutral operators `&+`, `&-`, `&*`, `&/`, `&^`, and parentheses `( )`.

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 object against the types `integer` and `symbol`. Remember that braces in a structured type mean that either type can be matched.

```
> type(5,{integer,symbol});
```

true **(11.67)**

```
> type(a,{integer,symbol});
```

true **(11.68)**

```
> type((2 &* 3) &+ 4,{integer,symbol});
```

false **(11.69)**

The above statements demonstrate that the type command can be used to distinguish between a single integer or variable and a complex expression.

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 Maple graphs, each vertex must be unique and distinct from all other vertices. In order to make Maple consider two 3s or two `&+`'s to be different, we define the following procedure.

```
> Unique := proc(a)
    convert(a,`local`);
  end proc:
```

The command `convert(a,`local`)` has the effect of turning the expression `a` into a local name. This means that the object returned is, as far as Maple is concerned, different from another copy of itself. For example, we can make 2 not equal to 2.

```
> evalb(Unique(2)=2);
```

false **(11.70)**

Identifying the operator and the operands
In the recursive case, we must separate a complex expression into its operator and the left and right

operands. Consider the following example.

```
> exampExpr := (3 &* 4) &+ (7 &* (3 &+ x));
```

$$exampExpr := (3 \;\&*\; 4) \;\&+\; (7 \;\&*\; 3 \;\&+\; x) \qquad \textbf{(11.71)}$$

This expression, $3\cdot 4 + 7\cdot(3 + x)$ consists of the sum of $3\cdot 4$ and $7\cdot(3 + x)$. In Maple, we will use the op command to separate the expression into its parts.

Recall that op has several forms. Typically we've used the form that accepts only one argument, typically a list or a set, and returns the sequence underlying the argument. Observe what happens if we apply op to our expression.

```
> op(exampExpr);
```

$$3 \;\&*\; 4,\ 7 \;\&*\; 3 \;\&+\; x \qquad \textbf{(11.72)}$$

It has removed the central addition and produced the sequence consisting of the two operands.

A different form of op will allow us to access the two operands directly. Given a positive integer as a first argument, op returns the specified operand.

```
> op(1,exampExpr);
```

$$3 \;\&*\; 4 \qquad \textbf{(11.73)}$$

```
> op(2,exampExpr);
```

$$7 \;\&*\; 3 \;\&+\; x \qquad \textbf{(11.74)}$$

Giving 0 as the first argument to op has slightly different meanings in different contexts. In this case, giving 0 as the first argument will return the operator.

```
> op(0,exampExpr);
```

$$\&+ \qquad \textbf{(11.75)}$$

Combining subtrees

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

```
> JoinTrees := proc(newR,A::BTree,B::BTree)
    local newT, newVerts, Aroot, Broot, newEdges, v, e, p, w;
    uses GraphTheory;
    newVerts := [newR,op(Vertices(A)),op(Vertices(B))];
    Aroot := GetGraphAttribute(A,"root");
    Broot := GetGraphAttribute(B,"root");
    newEdges := Edges(A) union Edges(B)
                         union {[newR,Aroot],[newR,Broot]};
    newT := Graph(newVerts,newEdges);
    for v in Vertices(A) do
      p := GetVertexAttribute(A,v,"order");
      SetVertexAttribute(newT,v,"order"=p);
    end do;
    for v in Vertices(B) do
      p := GetVertexAttribute(B,v,"order");
      SetVertexAttribute(newT,v,"order"=p);
    end do;
    SetVertexAttribute(newT,Aroot,"order"=1);
    SetVertexAttribute(newT,Broot,"order"=2);
    SetVertexAttribute(newT,newR,"order"=0);
    SetGraphAttribute(newT,"root"=newR);
    return newT;
```

```
end proc:
```

The procedure

With the **JoinTrees** procedure above and the **NewBTree** from Section 2, we are ready to write the procedure for turning infix expressions into binary trees.

The procedure accepts a single argument, **e**, the expression. We first test the type of **e**. If it is an integer or a symbol, then we use **NewBTree** to create a new binary tree with **e** as the only vertex. Note that we apply **Unique** to **e** before passing it to **NewBTree** to ensure that all vertices in the final tree are distinct.

Otherwise, we're in the recursive case. We use **op** to determine the left and right operands and the operator. Again, we apply **Unique** to the operator, as it will be a vertex in the tree. After recursive calls to the procedure to create the trees for the two operands, the subtrees are joined at the operator into the result tree.

Here is the procedure.

```
> InfixToTree := proc(e)
    local lhs, rhs, o, lhsTree, rhsTree, result;
    uses GraphTheory;
    if type(e,{integer,symbol}) then
      result := NewBTree(Unique(e));
    else
      lhs := op(1,e);
      o := Unique(op(0,e));
      rhs := op(2,e);
      lhsTree := InfixToTree(lhs);
      rhsTree := InfixToTree(rhs);
      result := JoinTrees(o,lhsTree,rhsTree);
    end if;
    return result;
  end proc:
```

We test the procedure on the example expression.

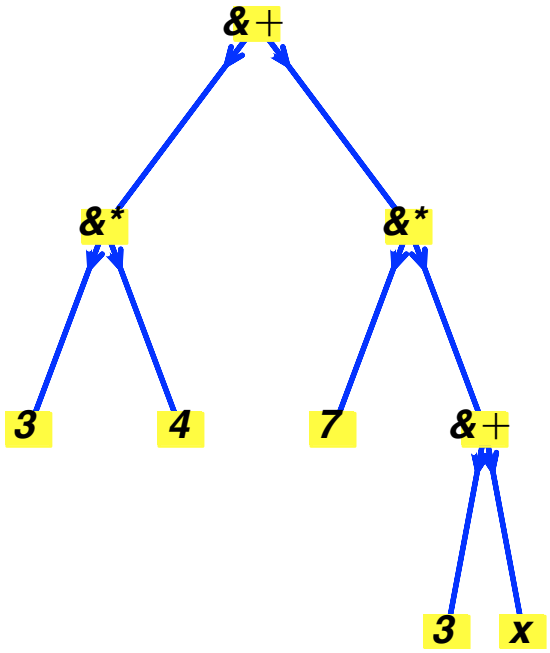
```
> exampExpr;
```

$$(3 \;\&*\; 4) \;\&+\; (7 \;\&*\; 3 \;\&+\; x) \qquad \textbf{(11.76)}$$

```
> exampTree := InfixToTree(exampExpr);
```

exampTree := Graph 26: a directed unweighted graph with 9 vertices and 8 arc(s) **(11.77)**

```
> DrawBTree(exampTree);
```



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. For example, applying `Inorder` to the tree we created in the last example yields the correct sequence of symbols, with the exception of omitted parentheses.

```
> Inorder(exampTree);
`3` `&*` `4` `&+` `7` `&*` `3` `&+` x
```

It is left to the reader to make the needed modifications to produce procedures that return accurate infix, prefix, and postfix expressions.

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.

```
> EvalPostfix := proc(Expr::list)
    local i, L;
    L := Expr;
    while nops(L) > 1 do
      i := 1;
      while not L[i] in {"+","-","*","/","^"} do
        i := i + 1;
      end do;
      if L[i] = "+" then
        L[i] := L[i-2] + L[i-1];
      elif L[i] = "-" then
        L[i] := L[i-2] - L[i-1];
      elif L[i] = "*" then
        L[i] := L[i-2] * L[i-1];
      elif L[i] = "/" then
```

```
        L[i] := L[i-2] / L[i-1];
      elif L[i] = "^" then
        L[i] := L[i-2] ^ L[i-1];
      end if;
      L := subsop(i-1=NULL,i-2=NULL,L);
    end do;
    return L[1];
  end proc:
> Post1 := [7,2,3,"*","-",4,"^",9,3,"/","+"];
```

$$Post1 := [7, 2, 3, "*", "-", 4, "^", 9, 3, "/", "+"] \quad \textbf{(11.78)}$$

```
> EvalPostfix(Post1);
```

$$4 \quad \textbf{(11.79)}$$

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

11.4 Spanning Trees

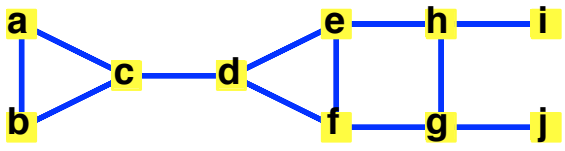
This section explains how to use Maple to construct spanning trees for graphs and how to use spanning trees to solve many different types of problems. Spanning trees have a myriad of applications, including coloring graphs, placing n queens on a $n \times n$ chessboard so that no two of the queens attack 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 Maple to form spanning trees using two algorithms: depth-first search and breadth-first search. Then we will show how to use Maple to solve the problems just mentioned.

Maple includes a command, **SpanningTree**, for finding a spanning tree of an undirected graph. To illustrate, we reproduce the graph from Exercise 13 of Section 11.4.

```
> Exercise13 := Graph({{"a","b"},{"a","c"},{"b","c"},{"c","d"},
  {"d","e"},{"d","f"},{"e","f"},{"e","h"},{"f","g"},{"g","h"},
  {"g","j"},{"h","i"}});
```

Exercise13 := Graph 27: an undirected unweighted graph with 10 vertices and 12 edge(s) **(11.80)**

```
> SetVertexPositions(Exercise13,[[0,1],[0,0],[1,.5],[2,.5],[3,
  1],[3,0],[4,0],[4,1],[5,1],[5,0]]);
> DrawGraph(Exercise13);
```



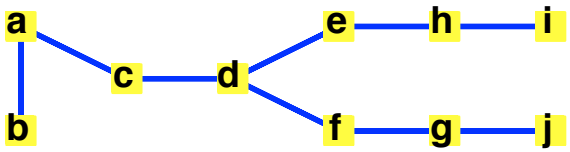
Now we use the **SpanningTree** command to find a spanning tree for this graph. The only required argument is the name of the graph whose spanning tree we wish to compute.

```
> Span1Exercise13 := SpanningTree(Exercise13);
```

Span1Exercise13 := **(11.81)**

Graph 28: an undirected unweighted graph with 10 vertices and 9 edge(s)

```
> DrawGraph(Span1Exercise13);
```



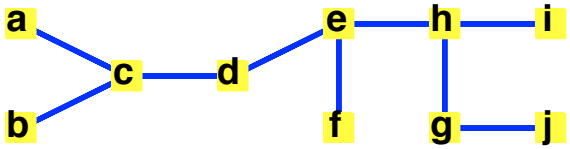
We can also specify one of the vertices of the graph as a second argument in order to specify the root of the spanning tree.

```
> Span2Exercise13 := SpanningTree(Exercise13,"i");
```

Span2Exercise13 := **(11.82)**

Graph 29: an undirected unweighted graph with 10 vertices and 9 edge(s)

```
> DrawGraph(Span2Exercise13);
```



Despite Maple's existing **SpanningTree** command, we will develop two of our own using depth-first and breadth-first search algorithms 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 the procedure terminates.

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

1. First, we check that the graph is connected using Maple's **IsConnected** procedure. If not, there can be no spanning tree and the procedure returns **FAIL**.
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.
   b) **Exploring** will be the list of vertices that are currently being explored. As vertices are visited, they are added the 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. Note that **T** will be neither rooted nor ordered.
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 set of 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,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 procedure.

```
> DepthSearch := proc(G::Graph, startV)
    local ToVisit, Exploring, T, v, N, w;
    uses GraphTheory;
    if not IsConnected(G) then
      return FAIL;
    end if;
    ToVisit := {op(Vertices(G))};
    Exploring := [startV];
    T := Graph(Vertices(G));
    while Exploring <> [] do
      v := Exploring[-1];
```

```
      N := {op(Neighbors(G,v))} intersect ToVisit;
      if N <> {} then
        w := N[1];
        AddEdge(T,{v,w});
        ToVisit := ToVisit minus {w};
        Exploring := [op(Exploring),w];
      else
        Exploring := subsop(-1=NULL,Exploring);
      end if;
    end do;
    return T;
  end proc:
```

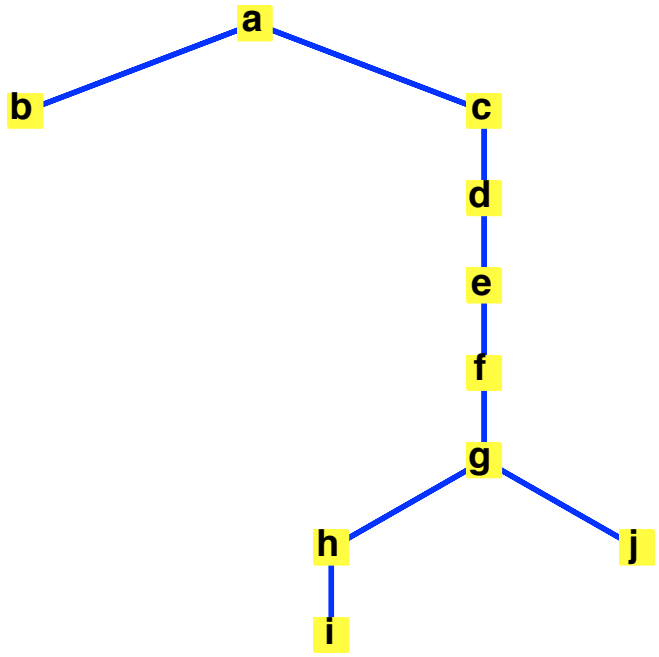
Let's test this with our Exercise 13 example from above.

```
> DepthExercise13 := DepthSearch(Exercise13,"a");
```

DepthExercise13 := **(11.83)**

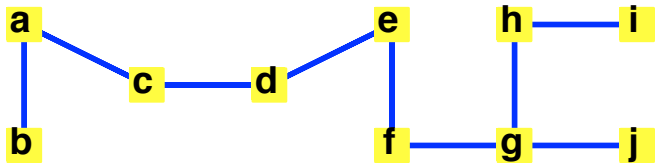
Graph 30: an undirected unweighted graph with 10 vertices and 9 edge(s)

```
> DrawGraph(DepthExercise13);
```



We can reposition the vertices to match the original graph with SetVertexPositions and GetVertexPositions.

```
> SetVertexPositions(DepthExercise13,GetVertexPositions
  (Exercise13));
> DrawGraph(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 procedure, 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 Maple's **IsConnected** procedure.
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 set of 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.

```
> BreadthSearch := proc(G::Graph, startV)
    local ToVisit, ToProcess, T, v, N, w;
    uses GraphTheory;
    if not IsConnected(G) then
```

```
      return FAIL;
    end if;
    ToVisit := {op(Vertices(G))} minus {startV};
    ToProcess := [startV];
    T := Graph(Vertices(G));
    while ToProcess <> [] do
      v := ToProcess[1];
      N := {op(Neighbors(G,v))} intersect ToVisit;
      for w in N do
        AddEdge(T,{v,w});
        ToProcess := [op(ToProcess),w];
        ToVisit := ToVisit minus {w};
      end do;
      ToProcess := subsop(1=NULL,ToProcess);
    end do;
    return T;
  end proc:
```

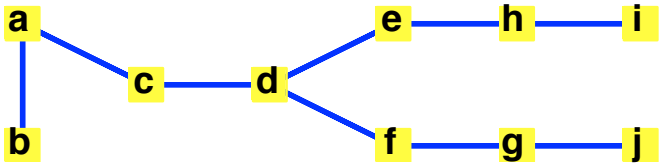
Once again, we illustrate using Exercise 13.

```
> BreadthExercise13 := BreadthSearch(Exercise13,"a");
```

BreadthExercise13 := **(11.84)**

Graph 31: an undirected unweighted graph with 10 vertices and 9 edge(s)

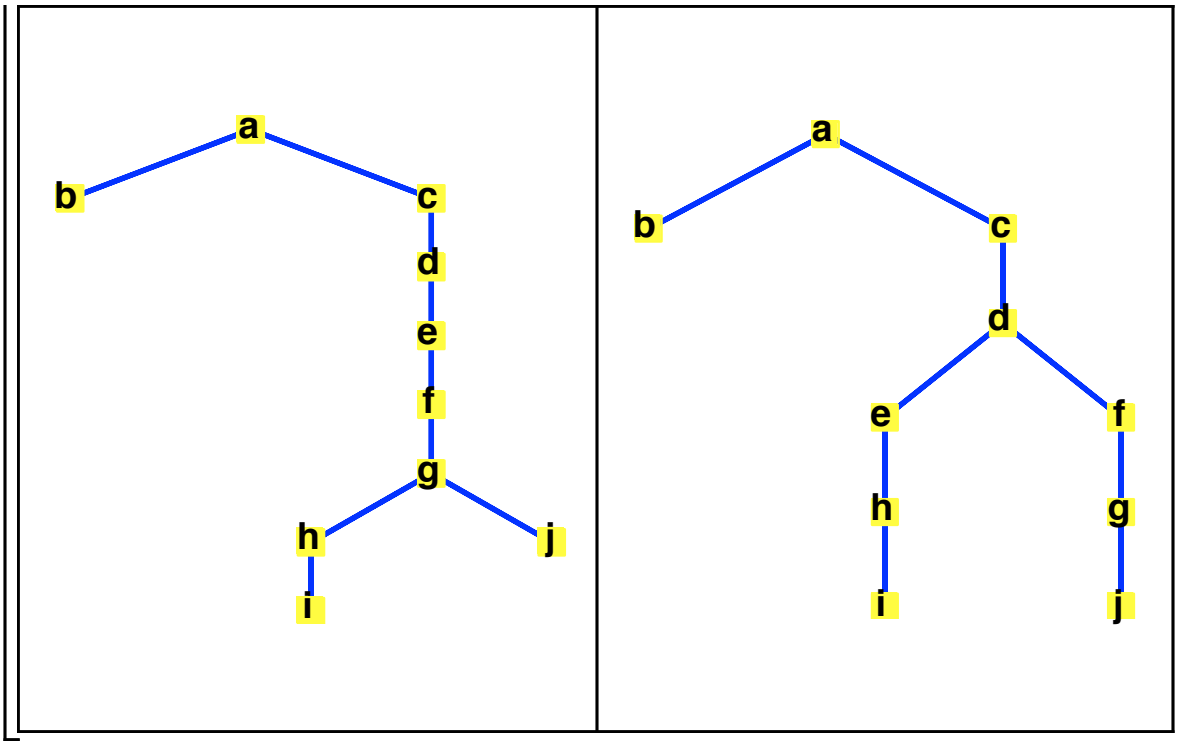
```
> SetVertexPositions(BreadthExercise13,GetVertexPositions
  (Exercise13));
> DrawGraph(BreadthExercise13);
```



Observe that this is the same graph as was produced by Maple's **SpanningTree** command, which suggests that Maple's command uses a breadth-first algorithm.

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

```
> DrawGraph([DepthExercise13,BreadthExercise13],style=tree,
  root="a");
```



Notice that the two spanning trees are quite different, even though they are both drawn rooted at vertex *a*. 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 $v_1, v_2, \ldots, v_m$ and fix an ordering of the colors as color 1, color 2, …, color *n*. We will use the ordering of the vertices that Maple automatically imposes. For the colors, we will require an ordered list of colors as one of the arguments to the procedure.
2. We store the current state of the coloring in a list we will call `coloring`. This *i*th entry in this list will correspond to the color of *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 procedure terminates in one of two cases. If **i** is set to a value greater then 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 procedure 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 using **HighlightVertex**. If it determines that there is no *n*-coloring of the graph, it will return **FAIL**.

```
> BackColor := proc(G::Graph, C::list)
    local Verts, numverts, allcolorsL, k, coloring, i, N, j,
  used, available, colorL;
    uses GraphTheory;
    Verts := Vertices(G);
    numverts := nops(Verts);
    allcolorsL := {seq(k,k=1..nops(C))};
    coloring := [1];
    i := 2;
    while i > 1 and i <= numverts do
      N := Neighbors(G,Verts[i]);
      used := {};
      for j from 1 to i-1 do
        if Verts[j] in N then
          used := used union {coloring[j]};
        end if;
      end do;
      if nops(coloring) >= i then
        used := used union {seq(k,k=1..coloring[i])};
      end if;
      available := allcolorsL minus used;
      if available <> {} then
        coloring := [op(coloring[1..i-1]),available[1]];
        i := i + 1;
      else
        if nops(coloring) >= i then
          coloring := coloring[1..(i-1)];
        end if;
        i := i - 1;
      end if;
    end do;
    if i > numverts then
      print(coloring);
      colorL := [];
      for k from 1 to numverts do
        colorL := [op(colorL),C[coloring[k]]];
```

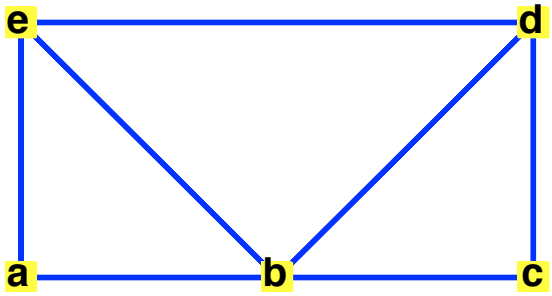
```
      end do;
      HighlightVertex(G,Verts,colorL);
      DrawGraph(G);
    else
      return FAIL;
    end if;
  end proc:
```

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

```
> Fig11Graph := Graph({{"a","b"},{"a","e"},{"b","c"},{"b","d"},
  {"b","e"},{"c","d"},{"d","e"}});
```

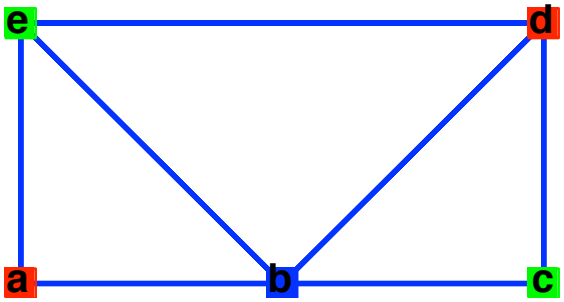
Fig11Graph := Graph 32: an undirected unweighted graph with 5 vertices and 7 edge(s) **(11.85)**

```
> SetVertexPositions(Fig11Graph,[[0,0],[1,0],[2,0],[2,1],[0,1]]
  );
> DrawGraph(Fig11Graph);
```




```
> BackColor(Fig11Graph,[red,blue,green]);
```

[1, 2, 3, 1, 3]



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

```
> K5 := CompleteGraph(5);
```

K5 := Graph 33: an undirected unweighted graph with 5 vertices and 10 edge(s) **(11.86)**

```
> BackColor(K5,[red,blue,green]);
```

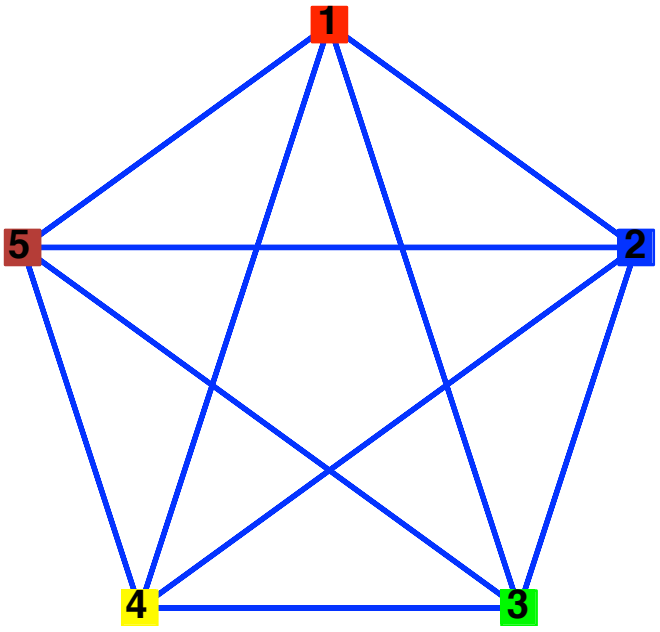
FAIL **(11.87)**

```
> BackColor(K5,[red,blue,green,yellow]);
```

FAIL **(11.88)**

```
> BackColor(K5,[red,blue,green,yellow,brown]);
```

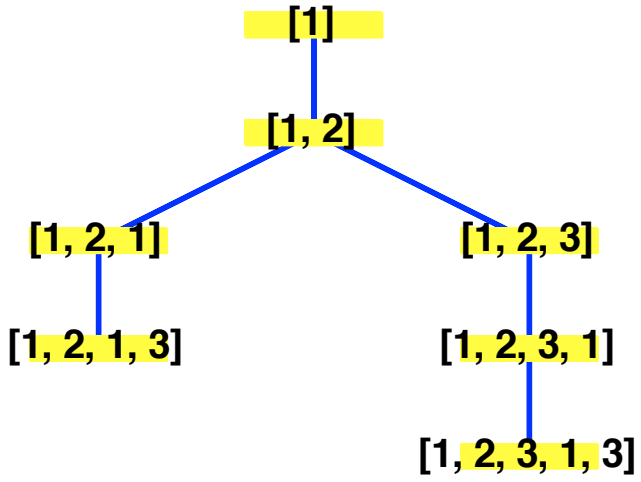
[1, 2, 3, 4, 5]



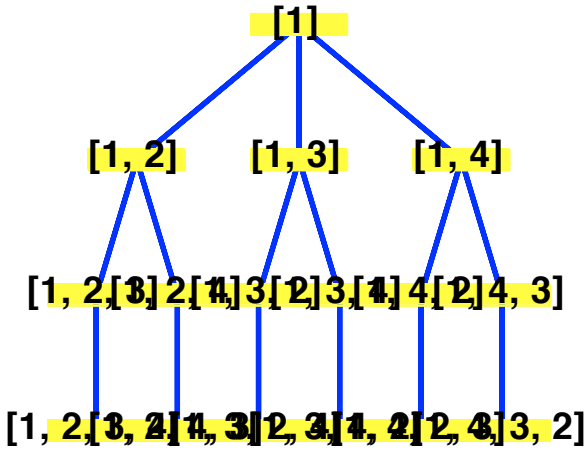
Before moving on to the *n*-queens problem, we illustrate how we can modify our backtracking procedure to record and display the decision tree. Instead of displaying the graph, our modified algorithm will produce an animation showing how the decision tree is built up. We do this by keeping a list of trees. Each time a color is assigned, we create a new tree in the list by adding the current state of the `coloring` list (converted to a string) as a vertex.

```
> BackColorDT := proc(G::Graph, C::list)
    local Verts, numverts, allcolorsL, k, coloring, i, N, j,
  used, available, colorL, DTList, parentV, newV, newT, Vpos,
  plotList;
    uses GraphTheory;
    Verts := Vertices(G);
    numverts := nops(Verts);
    allcolorsL := {seq(k,k=1..nops(C))};
    coloring := [1];
    newV := convert(coloring,`string`);
    DTList := [Graph([newV])];
    i := 2;
    while i > 1 and i <= numverts do
      N := Neighbors(G,Verts[i]);
      used := {};
      for j from 1 to i-1 do
```

```
        if Verts[j] in N then
          used := used union {coloring[j]};
        end if;
      end do;
      if nops(coloring) >= i then
        used := used union {seq(k,k=1..coloring[i])};
      end if;
      available := allcolorsL minus used;
      if available <> {} then
        parentV := convert(coloring[1..i-1],`string`);
        coloring := [op(coloring[1..i-1]),available[1]];
        newV := convert(coloring,`string`);
        newT := AddVertex(DTList[-1],newV);
        AddEdge(newT,{parentV,newV});
        DTList := [op(DTList),newT];
        i := i + 1;
      else
        if nops(coloring) >= i then
          coloring := coloring[1..(i-1)];
        end if;
        i := i - 1;
      end if;
    end do;
    DrawGraph(DTList):
    Vpos := GetVertexPositions(DTList[-1]);
    for k from 1 to nops(DTList) do
      SetVertexPositions(DTList[k],Vpos[1..k]);
    end do;
    plotList := [seq(DrawGraph(DTList[k]),k=1..nops(DTList))];
    plots[display](plotList,insequence=true);
  end proc:
> BackColorDT(Fig11Graph,[red,blue,green]);
```




```
> 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 procedure, `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. Note that we can create a square matrix with all entries initialized to 1 by issuing the `Matrix` command with two arguments: the dimension of the matrix and the formula `fill=1`. For example,

```
> Matrix(5,fill=1);
```

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix} \qquad \textbf{(11.89)}$$

initializes the matrix representing the board on which no queens have been placed.

We now build a **BoardStatus** procedure 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.

```
> BoardStatus := proc(curQueens, dim)
    local board, i, dif, Qrank, Qfile, vQueens;
    board := Matrix(dim, fill=1);
    for Qfile from 1 to nops(curQueens) do
      Qrank := curQueens[Qfile];
      for i from 1 to dim do
        board[Qrank,i] := 0;
        board[i,Qfile] := 0;
        dif := i - Qfile;
        if Qrank + dif <= dim and Qrank + dif >= 1 then
          board[Qrank+dif,i] := 0;
        end if;
        if Qrank - dif <= dim and Qrank - dif >= 1 then
          board[Qrank-dif,i] := 0;
        end if;
      end do;
    end do;
    for Qfile from 1 to nops(curQueens) do
      Qrank := curQueens[Qfile];
      board[Qrank,Qfile] := `Q`;
    end do;
    return board;
  end proc:
```

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.

```
> BoardStatus([2,7],10);
```

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\ Q & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\ 0 & Q & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} \quad \textbf{(11.90)}$$

The **ValidQueens** procedure 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** procedure and instead create the **ValidQueens** procedure independently, but, as you see above, the **BoardStatus**

procedure provides a useful visualization.

```
> ValidQueens := proc(curQueens,dim)
    local B, file, i, freeSet;
    B := BoardStatus(curQueens,dim);
    file := nops(curQueens) + 1;
    freeSet := {};
    for i from 1 to dim do
      if B[i,file] = 1 then
        freeSet := freeSet union {i};
      end if;
    end do;
    return freeSet;
  end proc:
```

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** procedure 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.

```
> nQueens := proc(boardDim::posint)
    local queens, file, open, i;
    queens := [];
    file := 1;
    while file > 0 and file <= boardDim do
      open := ValidQueens(queens[1..(file-1)],boardDim);
      if nops(queens) >= file then
        open := open minus {seq(i,i=1..queens[file])};
      end if;
      if open <> {} then
        queens := [op(queens[1..(file-1)]),open[1]];
        file := file + 1;
      else
        if nops(queens) >= file then
          queens := queens[1..(file-1)];
        end if;
        file := file - 1;
      end if;
    end do;
    if file > boardDim then
      return BoardStatus(queens,boardDim);
    else
      return FAIL;
```

```
    end if;
  end proc:
```

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

```
> nQueens(8);
```

$$\begin{bmatrix} Q & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & Q & 0 \\ 0 & 0 & 0 & 0 & Q & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & Q \\ 0 & Q & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & Q & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & Q & 0 & 0 \\ 0 & 0 & Q & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \textbf{(11.91)}$$

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 **add** command as follows.

```
> listofvalues := [3,7,11,15,-4];
```

$$listofvalues := [3, 7, 11, 15, -4] \quad \textbf{(11.92)}$$

```
> add(i,i=listofvalues);
```

$$32 \quad \textbf{(11.93)}$$

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 as follows.

```
> listofstuff := ["a","b","c","d","e","f","g"];
```

$$listofstuff := ["a", "b", "c", "d", "e", "f", "g"] \quad \textbf{(11.94)}$$

```
> listofindices := [1,3,4,7];
```

$$listofindices := [1, 3, 4, 7] \quad \textbf{(11.95)}$$

```
> listofstuff[listofindices];
```

$$["a", "c", "d", "g"] \quad \textbf{(11.96)}$$

```
> listofstuff[[3,4,6]];
```

$$["c", "d", "f"] \quad \textbf{(11.97)}$$

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

```
> SubSum := proc(S::set(integer),M::integer)
    local SList, Bindices, allIndices, i, availIndices, k,
  currSum;
    SList := [op(S)];
```

```
    Bindices := [];
    allIndices := {seq(k,k=1..nops(SList))};
    i := 1;
    currSum := 0;
    while i > 0 and currSum <> M do
      availIndices := allIndices minus {op(Bindices)};
      if nops(Bindices) >= i then
        availIndices := availIndices
                              minus {seq(k,k=1..Bindices[i])};
      end if;
      for k in availIndices do
         if currSum + SList[k] > M then
           availIndices := availIndices minus {k};
         end if;
      end do;
      if availIndices <> {} then
        Bindices := [op(Bindices[1..(i-1)]),availIndices[1]];
        i := i + 1;
      else
        if nops(Bindices) >= i then
          Bindices := Bindices[1..(i-1)];
        end if;
        i := i - 1;
      end if;
      currSum := add(k,k=SList[Bindices[1..(i-1)]]);
    end do;
    if i = 0 then
      return FAIL;
    else
      return SList[Bindices];
    end if;
  end proc:
> SubSum({31,27,15,11,7,5},39);
                              [5, 7, 27]                    (11.98)
> SubSum({31,27,15,11,7,5},40);
                                FAIL                        (11.99)
```

11.5 Minimum Spanning Trees

This section explains how to use Maple 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. While Maple includes commands, `PrimsAlgorithm` and `KruskalsAlgorithm`, that implement these algorithms, this is another case in which understanding the implementation can help you better understand the algorithms. We will develop our own procedures that implement these two algorithms rather than using Maple's.

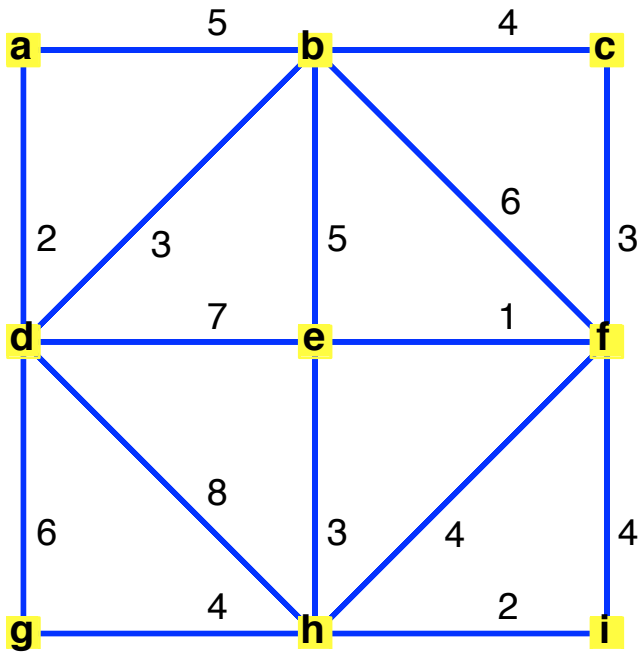
First, we we construct a graph to use as an example. We will recreate Exercise 3 from Section 11.5. Recall the syntax for defining undirected, weighted edges: the members of the edge set passed to the `Graph` command are two-element lists whose first element is the undirected edge and whose second element is the weight. For example `[{"a","b"},5]` represents an edge between a and b with weight 5.

```
> Exercise3 := Graph({[{"a","b"},5],[{"a","d"},2],[{"b","c"},
  4],[{"b","d"},3],[{"b","e"},5],[{"b","f"},6],[{"c","f"},3],[
  {"d","e"},7],[{"d","g"},6],[{"d","h"},8],[{"e","f"},1],[{"e",
  "h"},3],[{"f","h"},4],[{"f","i"},4],[{"g","h"},4],[{"h","i"},
  2]});
```

Exercise3 := Graph 34: an undirected weighted graph with 9 vertices and 16 edge(s) **(11.100)**

```
> SetVertexPositions(Exercise3,[[0,2],[1,2],[2,2],[0,1],[1,1],
  [2,1],[0,0],[1,0],[2,0]]);
```

```
> DrawGraph(Exercise3);
```

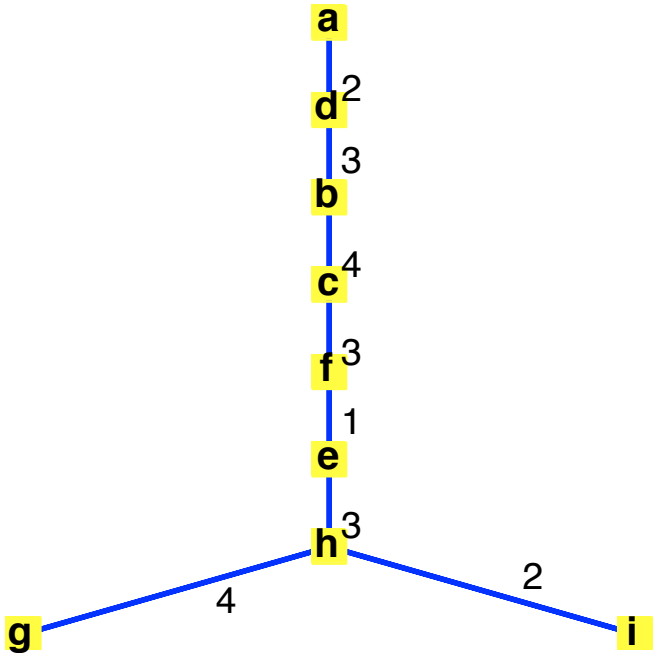


Before implementing our own versions of Prim's algorithm and Kruskal's algorithm, we'll mention how to use Maple's built-in implementations of these algorithms, **PrimsAlgorithm** and **KruskalsAlgorithm**. There is also a command called **MinimalSpanningTree**, which uses Kruskal's algorithm. The syntax for both commands is the same. There is one required argument, the name of the graph. The commands both return a graph object that is a minimum spanning tree for the input. We will use **PrimsAlgorithm** to illustrate, but recall that **KruskalsAlgorithm** uses the same syntax.

```
> Exercise3Prim := PrimsAlgorithm(Exercise3);
```

Exercise3Prim := Graph 35: an undirected weighted graph with 9 vertices and 8 edge(s) **(11.101)**

```
> DrawGraph(Exercise3Prim);
```



Both commands have two optional arguments. The first is a variable name, which will be assigned the weight of the minimal spanning tree.

```
> PrimsAlgorithm(Exercise3,'i');
```

Graph 36: an undirected weighted graph with 9 vertices and 8 edge(s) **(11.102)**

```
> i;
```

22 **(11.103)**

The second optional argument is the keyword **animate**. If this argument is given, the procedures will return an animation showing how the minimal spanning tree is built. Note, however, that this command will not work if strings are used to name vertices. Instead, we must use either numbers or variable names that have not been assigned values.

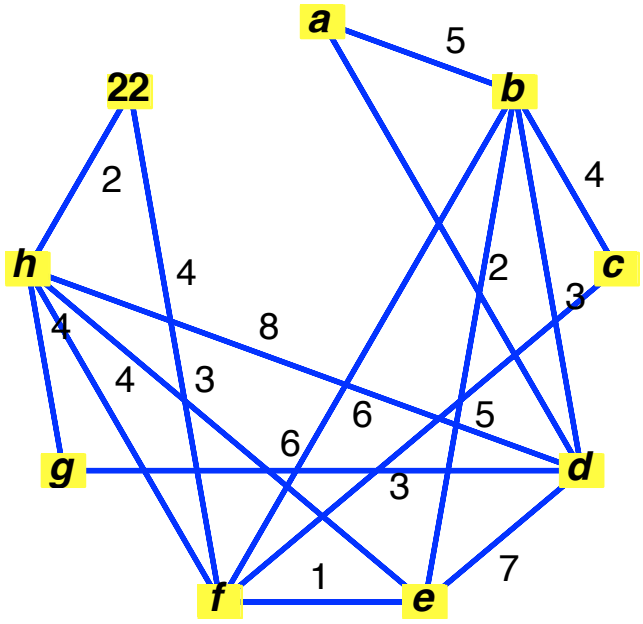
Below we define **Exercise3names**, which is identical to **Exercise3** but using names instead of strings. Note that the name **i** already stores a value. Even putting it in left single quotes to delay evaluation does not prevent it from appearing as its numerical value in the drawing of the graph.

```
> Exercise3names := Graph({[{a,b},5],[{a,d},2],[{b,c},4],[{b,d}
  ,3],[{b,e},5],[{b,f},6],[{c,f},3],[{d,e},7],[{d,g},6],[{d,h},
  8],[{e,f},1],[{e,h},3],[{f,h},4],[{f,'i'},4],[{g,h},4],[{h,
  'i'},2]});
```

Exercise3names := **(11.104)**

Graph 37: an undirected weighted graph with 9 vertices and 16 edge(s)

```
> DrawGraph(Exercise3names);
```

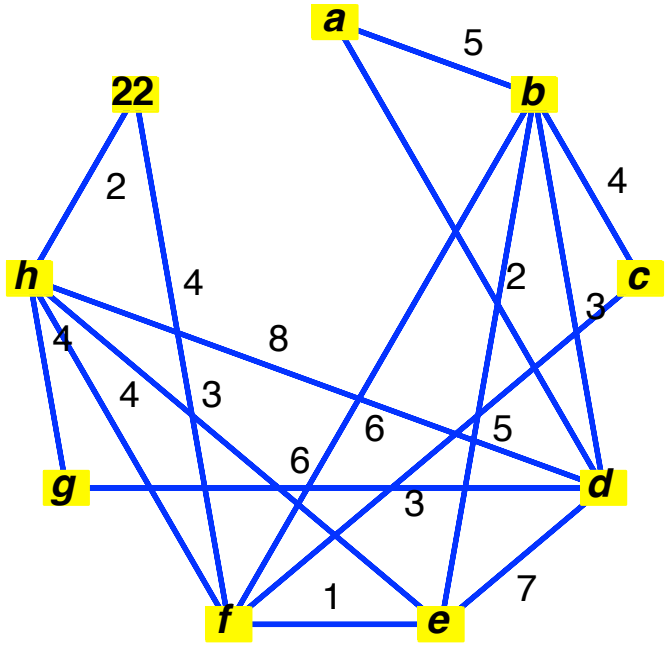


This is why, elsewhere in this chapter, we always use numbers or strings to name vertices in graphs. Note that it is also not possible to use the **animate** option after setting vertex positions with **SetVertexPositions**.

But we are now able to apply **PrimsAlgorithm** with the **animate** option.

```
> PrimsAlgorithm(Exercise3names,animate);
```

Input Graph



Prims algorithm builds a Minimum Cost Spanning Tree from a given starting vertex by adding to itself a 'fringe' edge of minimal weight at each iteration. If the edge would create a cycle in the tree it is discarded.

The first frame of the animation shows the graph. Subsequent frames show the process of building the minimal spanning tree one edge at a time.

Prim's Algorithm

We will now build our own versions of both algorithms. We will also see how to create animations that illustrate the process of building the spanning trees, but without the restrictions imposed by the built-in commands.

Since both algorithms 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 procedure that takes 2 arguments and returns true if the first argument is "less than" the second. As we've seen before, this procedure needs to also depend on the graph, so we will create a functional operator that takes a graph and produces the right kind of procedure that can be used by sort.

```
> edgeCompare := G  -> proc(a,b)
    local Wa, Wb;
    Wa := GetEdgeWeight(G,a);
    Wb := GetEdgeWeight(G,b);
    return evalb(Wa < Wb);
  end proc:
```

We will now consider Prim's algorithm, which is given as Algorithm 1 in Section 11.5 of the textbook. Prim's algorithm 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 procedure `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` determines the set of edges that are incident with a vertex currently in the tree using the `IncidentEdges` command. `IncidentEdges` takes two arguments. The first argument is a graph, and the second is either a vertex or a list of vertices. It returns the set of edges incident to the given vertex or vertices. The `MinEdge` procedure then eliminates any edge with both ends already in the spanning tree. (This is equivalent to the condition that the edge not introduce a simple circuit.) Once it has determined the valid candidates, the `MinEdge` procedure 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 procedure with the empty list as the second argument.

```
> MinEdge := proc(G::Graph, V::list)
    local possibleEdges, e, edgeList;
    uses GraphTheory;
    if V = [] then
      possibleEdges := Edges(G);
    else
      possibleEdges := IncidentEdges(G,V);
    end if;
    for e in possibleEdges do
      if e[1] in V and e[2] in V then
        possibleEdges := possibleEdges minus {e};
      end if;
    end do;
    if possibleEdges = {} then
      return NULL;
    end if;
```

```
    edgeList := [op(possibleEdges)];
    edgeList := sort(edgeList,edgeCompare(G));
    return edgeList[1];
  end proc:
```

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

1. Begin building the spanning tree by finding the edge of minimum weight with the command `MinEdge(G,[])`.
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 **AddEdge** 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 procedure by checking to see if the original graph has had its vertex positions explicitly set, and, if so, those positions are copied to the tree. (Note that we cannot conveniently use **SetVertexPositions** in this case, because the vertices in the spanning tree are likely in a different order than they are in the graph. Specifically, the order of the vertices in the spanning tree is the order in which they are added to the tree.)

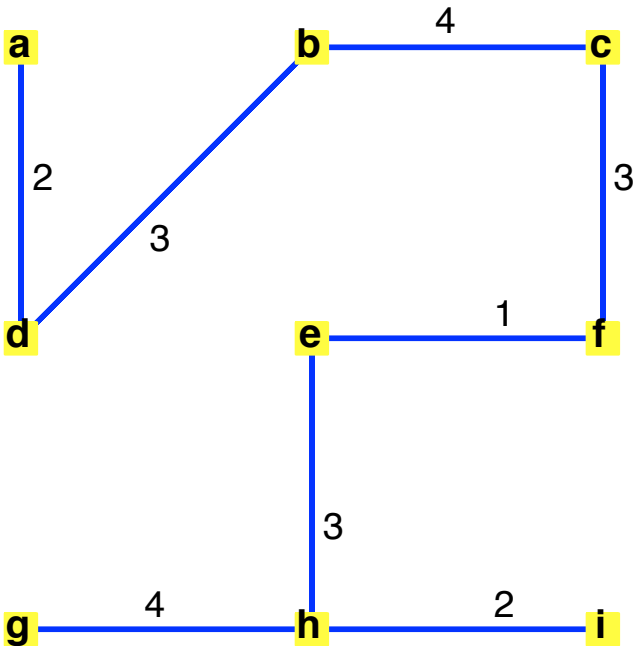
```
> Prim := proc(G::Graph)
    local newEdge, T, n, i, v, pos;
    uses GraphTheory;
    newEdge := MinEdge(G,[]);
    T := Graph(weighted,{newEdge});
    SetEdgeWeight(T,newEdge,GetEdgeWeight(G,newEdge));
    n := nops(Vertices(G));
    for i from 1 to n-2 do
      newEdge := MinEdge(G,Vertices(T));
      if newEdge[1] in Vertices(T) then
        T := AddVertex(T,newEdge[2]);
      else
        T := AddVertex(T,newEdge[1]);
      end if;
      AddEdge(T,newEdge);
      SetEdgeWeight(T,newEdge,GetEdgeWeight(G,newEdge));
    end do;
    for v in Vertices(T) do
      pos := GetVertexAttribute(G,v,"draw-pos-user");
      if pos <> FAIL then
        SetVertexAttribute(T,v,"draw-pos-user"=pos);
      end if;
    end do;
    return T;
  end proc:
```

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

```
> PrimExercise3 := Prim(Exercise3);
```

PrimExercise3 := Graph 38: an undirected weighted graph with 9 vertices and 8 edge(s) **(11.105)**

```
> DrawGraph(PrimExercise3);
```



Before moving on to Kruskal's algorithm, we'll create a procedure to produce an animation demonstrating Prim's algorithm in action. First, we'll modify the Prim procedure to record the list of edges that form the tree and return this list of edges rather than the tree.

```
> PrimEdges := proc(G::Graph)
    local newEdge, T, edgeList, n, i, v, pos;
    uses GraphTheory;
    newEdge := MinEdge(G,[]);
    T := Graph(weighted,{newEdge});
    SetEdgeWeight(T,newEdge,GetEdgeWeight(G,newEdge));
    edgeList := [newEdge];
    n := nops(Vertices(G));
    for i from 1 to n-2 do
      newEdge := MinEdge(G,Vertices(T));
      if newEdge[1] in Vertices(T) then
        T := AddVertex(T,newEdge[2]);
      else
        T := AddVertex(T,newEdge[1]);
      end if;
      AddEdge(T,newEdge);
      SetEdgeWeight(T,newEdge,GetEdgeWeight(G,newEdge));
      edgeList := [op(edgeList),newEdge];
    end do;
    return edgeList;
  end proc:
```

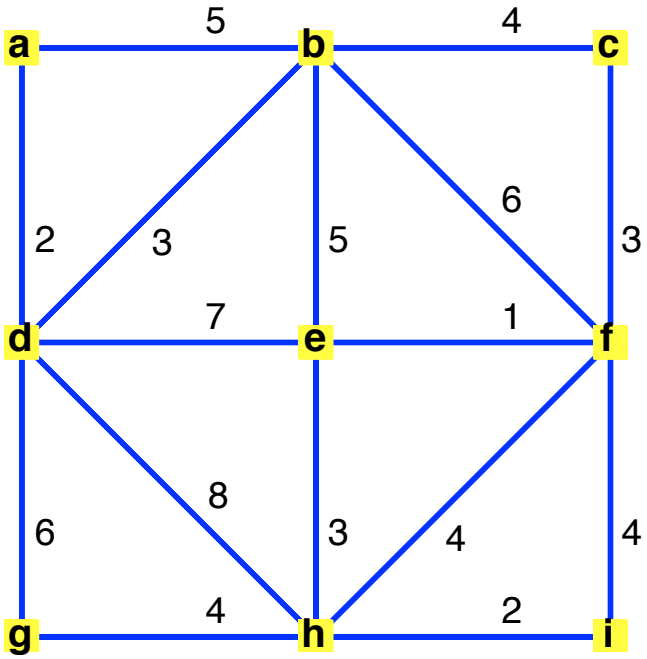
```
> PrimEdges(Exercise3);
```

[{"e", "f"}, {"e", "h"}, {"h", "i"}, {"c", "f"}, {"g", "h"}, {"b", "c"}, {"b", "d"}, {"a", "d"}] **(11.106)**

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

```
> plotTree := proc(G::Graph, T::list, n)
```

```
    local Gcopy, subtree, N, redlist;
    uses GraphTheory;
    Gcopy := CopyGraph(G);
    if n > nops(T) then
      N := nops(T);
    else
      N := floor(n);
    end if;
    if N <> 0 then
      subtree := T[1..N];
      redlist := [seq(red,i=1..N)];
      HighlightEdges(Gcopy,subtree,redlist);
    end if;
    DrawGraph(Gcopy);
  end proc:
> animateTree := proc(G::Graph, T::list)
    local t;
    plots[animate](plotTree, [G,T,t], t=0..(nops(T)),
                   paraminfo=false, frames=50);
  end proc:
> 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 procedure 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` command to find the trees. The `ConnectedComponents` command 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.

```
> NotFormsCircuit := proc(G::Graph,edge)
    local components, C;
    uses GraphTheory;
    components := ConnectedComponents(G);
    for C in components do
      if edge[1] in C and edge[2] in C then
        return false;
      end if;
    end do;
    return true;
  end proc:
```

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

1. Initialize `edges` to the list of edges of the given graph and sort this list using the `edgeCompare` procedure created above.
2. Initialize `T` to the empty graph.
3. Consider the first edge in the `edges` list. Use `NotFormsCircuit` 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 `NotFormsCircuit` 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.

```
> Kruskal := proc(G::Graph)
    local edges, T, n, i, newEdge, v, pos;
    uses GraphTheory;
    edges := [op(Edges(G))];
    edges := sort(edges,edgeCompare(G));
    T := Graph(weighted);
    n := nops(Vertices(G));
    i := 1;
    while i <= n-1 do
      newEdge := edges[1];
      if NotFormsCircuit(T,newEdge) then
        if not newEdge[1] in Vertices(T) then
          T := AddVertex(T,newEdge[1]);
        end if;
        if not newEdge[2] in Vertices(T) then
          T := AddVertex(T,newEdge[2]);
        end if;
        AddEdge(T,newEdge);
        SetEdgeWeight(T,newEdge,GetEdgeWeight(G,newEdge));
        i := i + 1;
      end if;
      edges := subsop(1=NULL,edges);
    end do;
    for v in Vertices(T) do
      pos := GetVertexAttribute(G,v,"draw-pos-user");
      if pos <> FAIL then
        SetVertexAttribute(T,v,"draw-pos-user"=pos);
      end if;
    end do;
```

```
    return T;
  end proc:
```

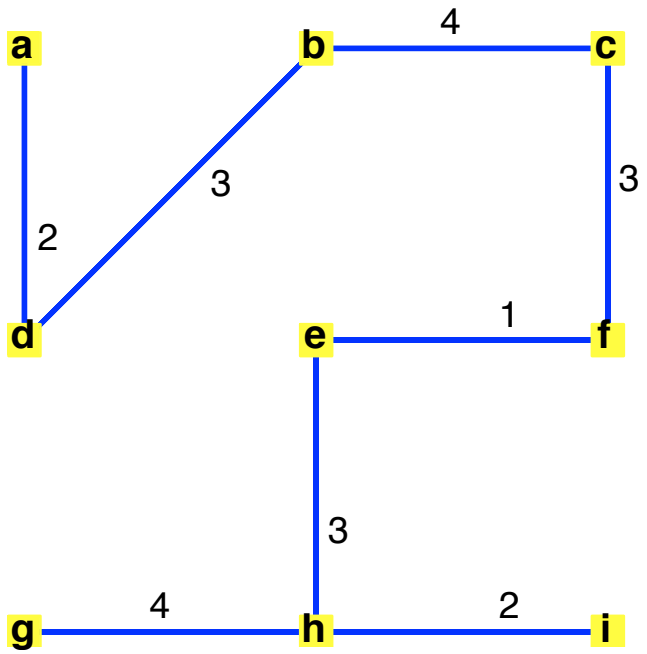
Note that Kruskal's algorithm produces the same minimum spanning tree for Exercise 3 as did our implementation of Prim's algorithm (in fact, this graph has a unique minimum spanning tree).

```
> KruskalExercise3 := Kruskal(Exercise3);
```

KruskalExercise3 := **(11.107)**

Graph 39: an undirected weighted graph with 9 vertices and 8 edge(s)

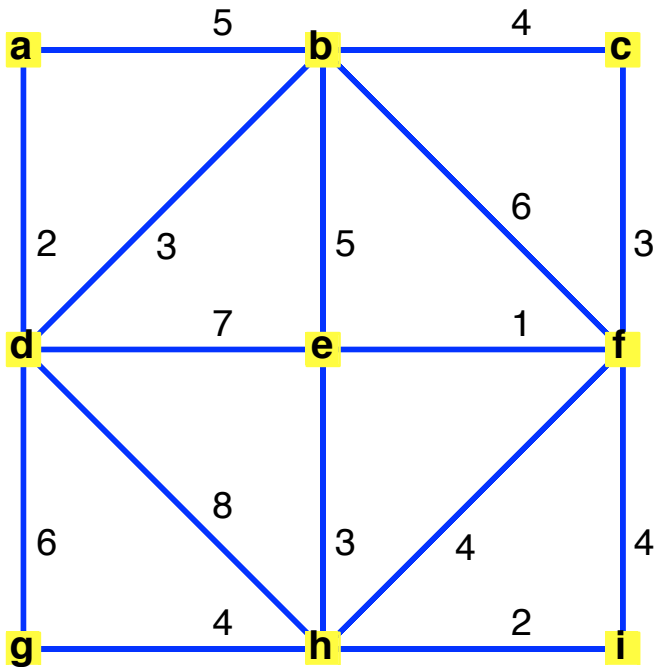
```
> DrawGraph(KruskalExercise3);
```



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

```
> KruskalEdges := proc(G::Graph)
    local edges, edgeList, T, n, i, newEdge;
    uses GraphTheory;
    edges := [op(Edges(G))];
    edges := sort(edges,edgeCompare(G));
    edgeList := [];
    T := Graph(weighted);
    n := nops(Vertices(G));
    i := 1;
    while i <= n-1 do
      newEdge := edges[1];
      if NotFormsCircuit(T,newEdge) then
        if not newEdge[1] in Vertices(T) then
          T := AddVertex(T,newEdge[1]);
        end if;
        if not newEdge[2] in Vertices(T) then
          T := AddVertex(T,newEdge[2]);
        end if;
        AddEdge(T,newEdge);
        SetEdgeWeight(T,newEdge,GetEdgeWeight(G,newEdge));
```

```
        edgeList := [op(edgeList),newEdge];
        i := i + 1;
      end if;
      edges := subsop(1=NULL,edges);
    end do;
    return edgeList;
  end proc:
> 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. This makes some of what follows slightly easier. The reader is encouraged to generalize the procedure we create here so as to not depend on this restriction.

We build the `ORTree` determined by the list of edges and add a "univ-address" attribute to each

vertex containing the vertex's universal address. First, here is an ordered list of edges for an ordered rooted tree.

```
> CP6Example := [["D","E"],["D","C"],["D","I"],["E","L"],
  ["E","F"],["I","B"],["I","K"],["I","H"],["F","J"],["F",
  "G"],["F","A"]];
```

CP6Example := [["D", "E"], ["D", "C"], ["D", "I"], ["E", "L"], ["E", "F"], ["I", "B"], ["I", "K"], ["I", "H"], ["F", "J"], ["F", "G"], ["F", "A"]] **(11.108)**

We have purposefully chosen vertex names that are out of order with respect to the ordering of the edges so that our construction of the **ORTree** 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 turning the list of edges into a set and passing it to the **Graph** command.

```
> CP6ExGraph := Graph({op(CP6Example)});
```

CP6ExGraph := **(11.109)**

Graph 40: a directed unweighted graph with 12 vertices and 11 arc(s)

Our first task is to turn this into an ordered rooted tree. Since we provided directed edges, all that is required to make the graph rooted is to set the graph attribute "root". Since the edges were ordered, the first edge is the edge from the root to the root's first child. This means that we can get the root by accessing the first element of the first edge.

```
> CP6ExRoot := CP6Example[1][1];
```

CP6ExRoot := "D" **(11.110)**

```
> SetGraphAttribute(CP6ExGraph,"root"=CP6ExRoot);
> type(CP6ExGraph,RTree);
```

true **(11.111)**

To make this graph an ordered rooted tree, we need to set the "order" attribute for each vertex. For the root, we just set the root's order to 0.

```
> SetVertexAttribute(CP6ExGraph,CP6ExRoot,"order"=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.

```
> curParent := CP6ExRoot:
  childOrder := 1:
  for thisEdge in CP6Example do
    if thisEdge[1] <> curParent then
      curParent := thisEdge[1];
      childOrder := 1;
    end if;
```

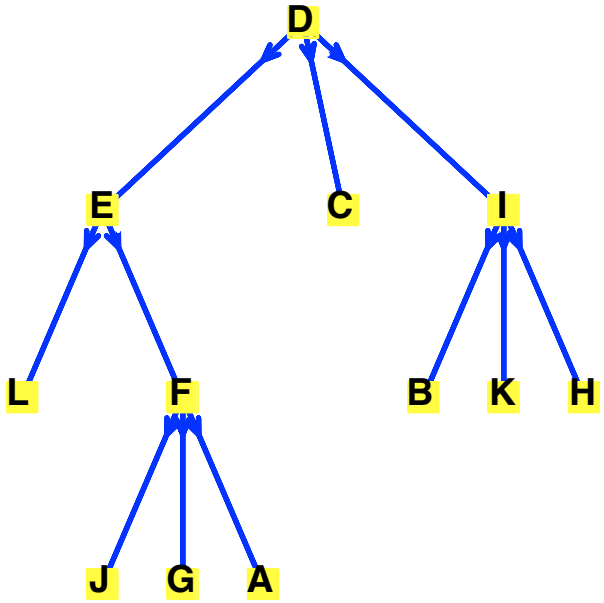
```
    SetVertexAttribute(CP6ExGraph,thisEdge[2],
                                  "order"=childOrder);
    childOrder := childOrder + 1;
  end do:
```

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

```
> type(CP6ExGraph,ORTree);
```

true **(11.112)**

```
> DrawORTree(CP6ExGraph);
```



Now the preliminaries are done and we can determine the universal addresses of the vertices. Again, the root will have universal address 0.

```
> SetVertexAttribute(CP6ExGraph,CP6ExRoot,"univ-address"="0")
  ;
```

The universal address of the root's children will be the same as their "order".

```
> for CPRootChild in FindChildren(CP6ExGraph,CP6ExRoot) do
    tempAddress := GetVertexAttribute(CP6ExGraph,CPRootChild,
                                                  "order");
    tempaddress := convert(tempAddress,`string`);
    SetVertexAttribute(CP6ExGraph,CPRootChild,
      "univ-address"=tempaddress);
  end do:
```

After that, we proceed as in a breadth-first search though the tree. We initialize a list with the children of the root.

```
> ToProcessAddress := FindChildren(CP6ExGraph,CP6ExRoot);
```

ToProcessAddress := ["C", "E", "I"] **(11.113)**

We process a vertex by (1) adding its children to the list; (2) setting the children's universal addresses to the result of concatenating the parent's address and the child's order; and (3) removing it from the list.

```
> while ToProcessAddress <> [] do
    processV := ToProcessAddress[1];
    Vaddress := GetVertexAttribute(CP6ExGraph,
                                   processV,"univ-address");
```

```
    for CToAddress in FindChildren(CP6ExGraph,processV) do
      ToProcessAddress := [op(ToProcessAddress),CToAddress];
      Caddress := GetVertexAttribute(CP6ExGraph,
                                     CToAddress,"order");
      Caddress := cat(Vaddress,".",Caddress);
      SetVertexAttribute(CP6ExGraph,CToAddress,
                                     "univ-address"=Caddress);
    end do:
    ToProcessAddress := subsop(1=NULL,ToProcessAddress);
  end do:
```

Now, we have set the universal address attribute for all the vertices. For example,

```
> GetVertexAttribute(CP6ExGraph,"A","univ-address");
                                "1.2.3"                          (11.114)
```

indicates that *A* is the root's first child's second child's third child. Or, *A* is the third child of the second child of the first child of the root.

We now put this together in a procedure.

```
> UniversalAddress := proc(edgeList::list)
    local T, root, curParent, childOrder, thisEdge, V,
      tempAddress, ToProcessAddress, Vaddress, C, Caddress;
    uses GraphTheory;
    T := Graph({op(edgeList)});
    root := edgeList[1][1];
    SetGraphAttribute(T,"root"=root);
    SetVertexAttribute(T,root,"order"=0);
    curParent := root;
    childOrder := 1;
    for thisEdge in edgeList do
      if thisEdge[1] <> curParent then
        curParent := thisEdge[1];
        childOrder := 1;
      end if;
      SetVertexAttribute(T,thisEdge[2],"order"=childOrder);
      childOrder := childOrder + 1;
    end do;
    SetVertexAttribute(T,root,"univ-address"="0");
    for V in FindChildren(T,root) do
      tempAddress := GetVertexAttribute(T,V,"order");
      tempAddress := convert(tempAddress,`string`);
      SetVertexAttribute(T,V,"univ-address"=tempAddress);
    end do;
    ToProcessAddress := FindChildren(T,root);
    while ToProcessAddress <> [] do
      V := ToProcessAddress[1];
      Vaddress := GetVertexAttribute(T,V,"univ-address");
      for C in FindChildren(T,V) do
        ToProcessAddress := [op(ToProcessAddress),C];
        Caddress := GetVertexAttribute(T,C,"order");
        Caddress := cat(Vaddress,".",Caddress);
        SetVertexAttribute(T,C,"univ-address"=Caddress);
      end do:
      ToProcessAddress := subsop(1=NULL,ToProcessAddress);
    end do:
    return T;
  end proc:
```

Computations and Explorations 1

Display all trees with six vertices.

Solution: To solve this problem, we make use of a recursive definition of tress. The empty graph is a tree and the graph with a single vertex is also 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 procedure, called **ExtendTrees**, that accepts as input a set of trees on n vertices and returns the resulting set 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.

```
> ExtendTrees := proc(Trees::set)
    local newTrees, newV, T, v, newT;
    uses GraphTheory;
    newTrees := {};
    newV := nops(Vertices(Trees[1])) + 1;
    for T in Trees do
      for v in Vertices(T) do
        newT := AddVertex(T,newV);
        AddEdge(newT,{v,newV});
        newTrees := newTrees union {newT};
      end do;
    end do;
    return newTrees;
  end proc:
```

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.

```
> AllTrees := {Graph([1])};
```

AllTrees := {*Graph 41: a graph with 1 vertex and no edges*} **(11.115)**

```
> for i from 2 to 4 do
    AllTrees := ExtendTrees(AllTrees);
  end do;
```

AllTrees :=

{*Graph 42: an undirected unweighted graph with 2 vertices and 1 edge(s)*}

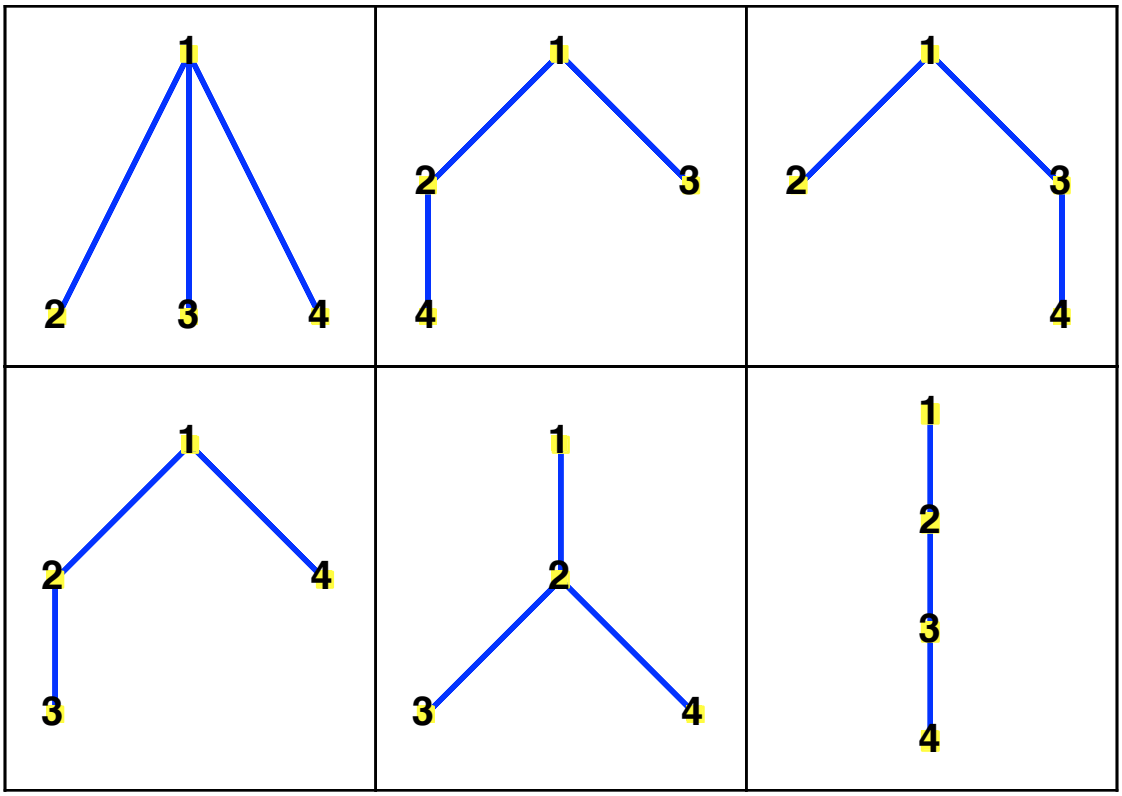
AllTrees :=

{*Graph 43: an undirected unweighted graph with 3 vertices and 2 edge(s)*,
Graph 44: an undirected unweighted graph with 3 vertices and 2 edge(s)}

AllTrees := **(11.116)**

{*Graph 45: an undirected unweighted graph with 4 vertices and 3 edge(s)*,
Graph 46: an undirected unweighted graph with 4 vertices and 3 edge(s),
Graph 47: an undirected unweighted graph with 4 vertices and 3 edge(s),
Graph 48: an undirected unweighted graph with 4 vertices and 3 edge(s),
Graph 49: an undirected unweighted graph with 4 vertices and 3 edge(s),
Graph 50: an undirected unweighted graph with 4 vertices and 3 edge(s)}

```
> DrawGraph(AllTrees,style=tree,root=1);
```



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 procedure, **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. (Note: you enter newline characters in a string in the same way as you create new lines when entering a procedure, by pressing the shift and enter/return keys simultaneously.)

```
> inputText := "The quick brown fox said,
  ""How do you do, my friend?""
  Then he ran very quickly off into the sunset.";
```

inputText := "The quick brown fox said, **(11.117)**

"How do you do, my friend?"

Then he ran very quickly off into the sunset."

Note that characters in a string can be accessed as if the string were a list. For instance, the fifth

character in the text above is

```
> inputText[5];
```

"q" **(11.118)**

Using **printf**, Maple will display the newline characters for us explicitly. Also, Maple differentiates between a newline character and a space character.

```
> printf(%a,inputText[26]);
"\n"
> printf(%a,inputText[4]);
" "
> evalb(inputText[4]=inputText[26]);
```

false **(11.119)**

We will calculate the frequencies of characters in our text by creating a table. We loop through the characters of the string. Recall that Maple treats strings much like a list of characters, and, in particular, we can use a for loop with the string in place of a list. For each character, we'll first check to see if the character is already a key of the table by attempting to access the value associated with the character and checking to see if the result is a number. If the table does have an entry, we will increment its value. If not, we will set the value associated to the character to 1. The result of this procedure is a table whose keys are the characters that appear in the string and whose associated values are the number of occurrences.

```
> FrequencyTable := proc(s::string)
    local T, c;
    T := table();
    for c in s do
      if type(T[c],posint) then
        T[c] := T[c] + 1;
      else
        T[c] := 1;
      end if;
    end do;
    return T;
  end proc:
```

We will apply this procedure to our example string. We can see that the input text includes 7 "e"s, 2 newline characters, and 17 spaces.

```
> inputTable := FrequencyTable(inputText);
```

inputTable := *T* **(11.120)**

```
> inputTable["e"];
```

7 **(11.121)**

```
> inputTable["\n"];
```

2 **(11.122)**

```
> inputTable[" "];
```

17 **(11.123)**

All that remains is to transform this data into the format expected by the Huffman algorithm: a list of pairs of the characters and the relative frequency of their occurrence. We use the **indices** command to obtain the list of all unique characters that appear in the string. We use the **length** command to find the total number of characters in the string.

```
> FrequencyList := proc(s::string)
```

```
    local Ftable, len, Flist, c, x;
    Ftable := FrequencyTable(s);
    len := length(s);
    Flist := [];
    for c in indices(Ftable,`nolist`) do
      x := [c,Ftable[c]/len];
      Flist := [op(Flist),x];
    end do;
    return Flist;
  end proc:
```

Finally, we're able to apply the **HuffmanCode** procedure.

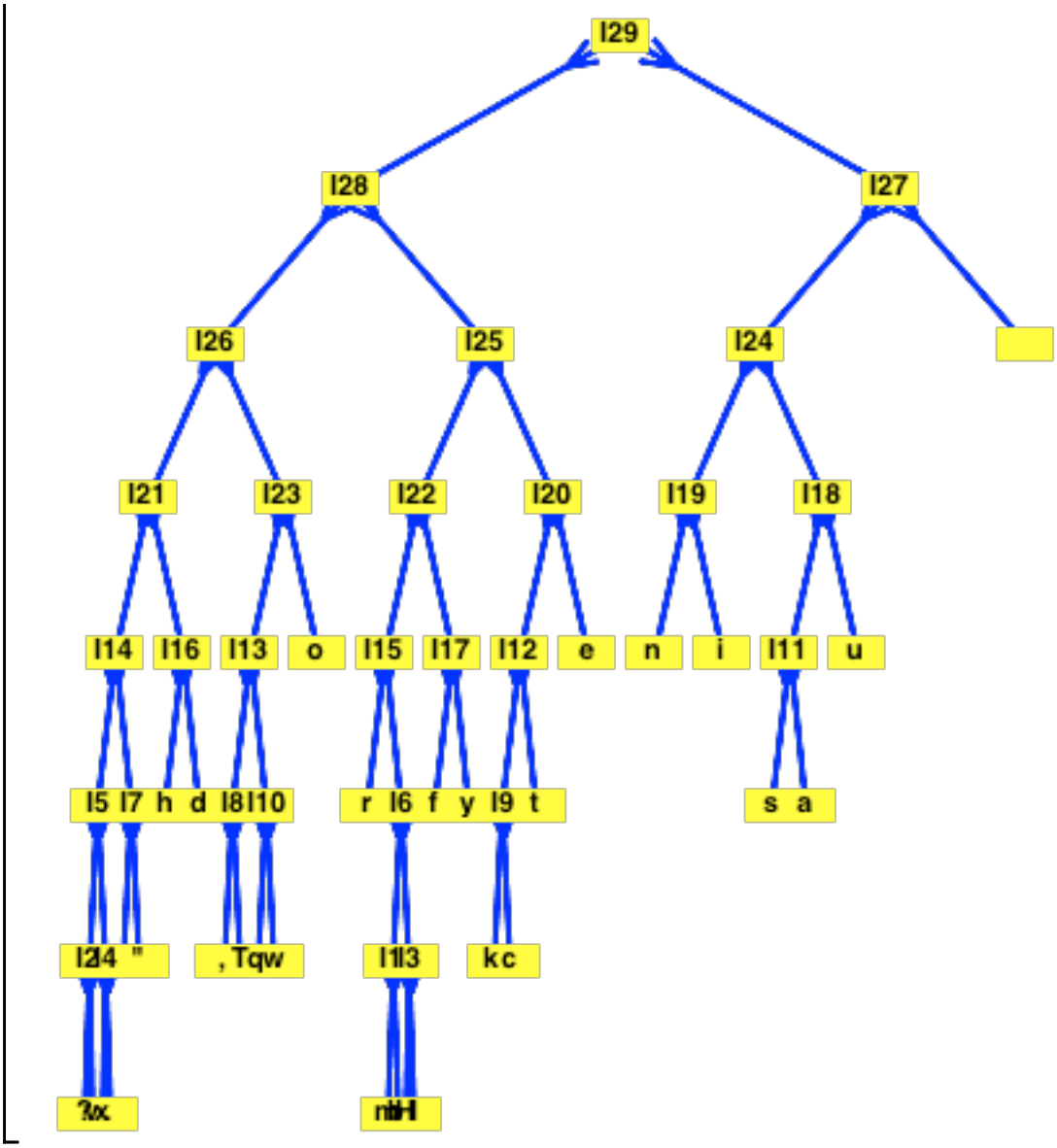
```
> foxList := FrequencyList(inputText);
```

$$foxList := \left[\left["v", \frac{1}{99}\right], \left["o", \frac{8}{99}\right], \left["T", \frac{2}{99}\right], \left["?", \frac{1}{99}\right], \left["n", \frac{2}{33}\right], \left["d", \frac{4}{99}\right], \left[" ", \frac{17}{99}\right], \left["h", \frac{4}{99}\right], \left[",", \frac{2}{99}\right], \left["w", \frac{2}{99}\right], \left["t", \frac{1}{33}\right], \left["q", \frac{2}{99}\right], \left[".", \frac{1}{99}\right], \left["u", \frac{4}{99}\right], \left["e", \frac{7}{99}\right], \left["i", \frac{5}{99}\right], \left["s", \frac{1}{33}\right], \left["a", \frac{2}{99}\right], \left["f", \frac{4}{99}\right], \left["k", \frac{2}{99}\right], \left["x", \frac{1}{99}\right], \left["H", \frac{1}{99}\right], \left["l", \frac{1}{99}\right], \left["y", \frac{4}{99}\right], \left["c", \frac{2}{99}\right], \left["r", \frac{4}{99}\right], \left["
", \frac{2}{99}\right], \left["m", \frac{1}{99}\right], \left["""", \frac{2}{99}\right], \left["b", \frac{1}{99}\right]\right] \quad (11.124)$$

```
> foxCode := HuffmanCode(foxList);
```

$$foxCode := \text{Graph 51: a directed weighted graph with 59 vertices and 58 arc(s)} \quad (11.125)$$

```
> DrawBTree(foxCode);
```



You will need to enlarge the image substantially in order to see the result in a readable format. Also note that Maple automatically suppresses the display of the edge weights for a tree this large.

Computations and Explorations 4

Compute the number of different spanning trees of K_n for $n = 1, 2, 3, 4, 5, 6$. Conjecture a formula for the number of such spanning trees whenever n is a positive integer.

Solution: Maple provides a command, `NumberOfSpanningTrees`, which returns the number of unique spanning trees of an undirected graph. So, the number of spanning trees on K_n for n from 1 to 6 can be computed as follows.

```
> seq(NumberOfSpanningTrees(CompleteGraph(n)),n=1..6);
```

$$1, 1, 3, 16, 125, 1296 \qquad \textbf{(11.126)}$$

We leave it to the reader to make a conjecture for the formula.

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 Maple procedure 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 procedure 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 procedure, 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.

```
> CheckersStart := Matrix([[0,2,0,2],[0,0,0,0],[0,0,0,0],[1,
  0,1,0]]);
```

$$CheckersStart := \begin{bmatrix} 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \qquad \textbf{(11.127)}$$

Given a matrix representing a board state and an integer representing which side's turn it is, the procedure **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 with `LinearAlgebra[Copy]` (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 procedure:

```
> MovePiece := proc(Board::Matrix,side::integer)
    local newBoards, direction, r, c, isKing, King, rowDir,
          colDir, newB;
    newBoards := [];
    if side = 1 then
      direction := -1;
    else
      direction := 1;
    end if;
    for r from 1 to 4 do
    for c from 1 to 4 do
    if abs(Board[r,c]) = side then
      if Board[r,c] < 0 then
        isKing := 1;
      else
        isKing := 0;
      end if;
      for King from 0 to isKing do
        if King = 0 then
          rowDir := direction;
        else
          rowDir := -1 * direction;
        end if;
        if r+rowDir >= 1 and r+rowDir <= 4 then
        for colDir from -1 to 1 by 2 do
        if c+colDir >= 1 and c+colDir <= 4 then
          if Board[r+rowDir,c+colDir] = 0 then
            newB := LinearAlgebra[Copy](Board);
            if ((r+rowDir=1 and side=1) or
                (r+rowDir=4 and side=2))
                  and Board[r,c] > 0 then
              newB[r+rowDir,c+colDir] := -1*Board[r,c];
            else
              newB[r+rowDir,c+colDir] := Board[r,c];
```

```
            end if;
            newB[r,c] := 0;
            newBoards := [op(newBoards),newB];
          elif abs(Board[r+rowDir,c+colDir]) <> side then
            if r+2*rowDir >= 1 and r+2*rowDir <= 4 and
               c+2*colDir >= 1 and c+2*colDir <= 4 then
            if Board[r+2*rowDir,c+2*colDir] = 0 then
              newB := LinearAlgebra[Copy](Board);
              if ((r+2*rowDir=1 and side=1) or
                  (r+2*rowDir=4 and side=2))
                    and Board[r,c] > 0 then
                newB[r+2*rowDir,c+2*colDir] := -1*Board[r,c];
              else
                newB[r+2*rowDir,c+2*colDir] := Board[r,c];
              end if;
              newB[r,c] := 0;
              newB[r+rowDir,c+colDir] := 0;
              newBoards := [op(newBoards),newB];
            end if;
            end if;
          end if;
        end if;
        end do;
        end if;
      end do;
    end if;
    end do;
    end do;
    return newBoards;
  end proc:
```

We now demonstrate a few steps using the procedure.

```
> Move1 := MovePiece(CheckersStart,1);
```

$$Move1 := \left[\begin{bmatrix} 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}\right] \quad \textbf{(11.128)}$$

```
> Move2 := MovePiece(Move1[1],2);
```

$$Move2 := \left[\begin{bmatrix} 0 & 0 & 0 & 2 \\ 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & 2 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\right] \quad \textbf{(11.129)}$$

```
> Move3 := MovePiece(Move2[3],1);
```

$$Move3 := \left[\begin{bmatrix} 0 & 2 & 0 & 0 \\ 1 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 2 & 0 & -1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}\right] \quad \textbf{(11.130)}$$

```
> Move4 := MovePiece(Move3[2],2);
```

$$Move4 := \left[\begin{bmatrix} 0 & 0 & 0 & -1 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 & 0 & -1 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}\right] \tag{11.131}$$

Exercises

Exercise 1. Write Maple procedures 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.)

Exercise 2. Develop a Maple procedure for constructing rooted Fibonacci trees. (See the prelude to Exercise 45 of Section 11.1 for a definition of a Fibonacci tree.)

Exercise 3. Develop a Maple procedure for listing the vertices of an ordered rooted tree in level order.

Exercise 4. Compare the performance of binary search trees to linear search as follows:

a) Write a procedure, `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 command `combinat[randperm](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` command to the list to create a binary search tree for the data.

c) Randomly select some positive integers to search for. The `randcomb` command could be useful here.

d) Use both `LinearSearch` and `BInsertion` to find the integers from part c in the list and tree, respectively. Time them using the typical `st := time(): procedure: time() - st;` structure. Repeat this for 100 different permutations and compare the resulting times. Compare these data (representing average-case complexity) with the theoretical worst-case results of n comparisons for `LinearSeach` and $\lceil \log(n+1) \rceil$ for `BInsertion`.

Exercise 5. Construct a Maple procedure 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` procedure, the algorithm should return the original string.

Exercise 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 Maple and make use of the procedures 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.

Exercise 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.

Exercise 8. Write a Maple procedure 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.)

Exercise 9. Explore the relative complexity of `Prim`, `Kruskal`, and the reverse-delete procedure you created in the previous exercise. Use the the `RandomGraph` command (available in the `RandomGraphs` subpackage of `GraphTheory`) to experiment with their performance. The command `RandomGraph(v,e,connected,weights=x..y)` will produce a random weighted connected graph with `v` vertices, `e` edges, and edge weights chosen randomly between `x` and `y` (with `x` < `y`). For each algorithm, can you find properties that you can impose on the graphs that will ensure that the algorithm will outperform the others?

Exercise 10. Develop a Maple procedure for producing degree-constrained spanning trees, which are defined in the Supplementary Exercises for Chapter 11. Use this procedure 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.

Exercise 11. Use Maple 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.)

Exercise 12. Use Maple 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.)

Exercise 13. Develop Maple procedures for finding a path through a maze using the technique of backtracking.

Exercise 14. Develop Maple procedures for solving Sudoku puzzles using the technique of backtracking.

Exercise 15. Use Maple 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.

Exercise 16. Alter the postfix expression evaluator, `EvalPostfix`, to handle prefix expressions.