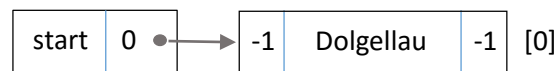


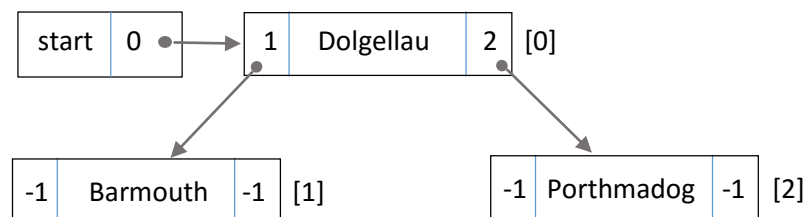
# 15 Binary trees

In this chapter we will examine a fourth abstract data structure, a **binary tree**. A set of data is kept in order in a binary tree by links between the data items in a similar way to a linked list. However, each record can be linked to **two other records**. As an example, suppose that a tourist information centre wishes to keep information of places of interest to visit in Wales. It should be possible to quickly find any record using the **town name** as the search value.

As in the linked list, a **start pointer** indicates the position of the first town. The first item in the binary tree is known as the **root node**.



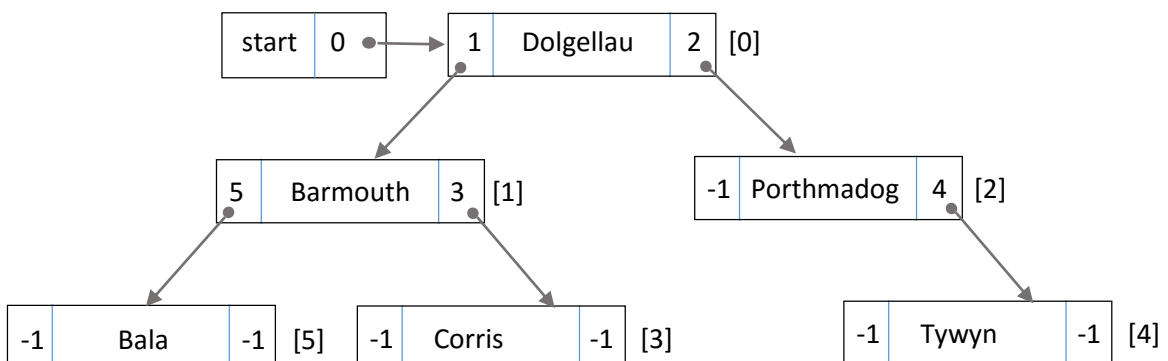
Further towns can be linked to either the **left pointer** or **right pointer** in the record. The general rule is that a data value earlier in the alphabet is attached as a left branch, and a data value later in the alphabet is attached as a right branch. We set pointers to **-1** to indicate the ends of branches.



To add further records, start at the **root node** then **branch left or right** at each node according to **alphabetical order** until the end of a branch is reached. The new record is attached at that point. For example, to add a record for **Corris**:

Start at the root node **Dolgellau**. C is before D, so branch left.  
 The node for **Barmouth** has been reached. C is after B, so branch right.  
 There is no right branch from **Barmouth** yet, so attach **Corris** at this point.

Using the same method, **Tywyn** would form a right branch from **Porthmadog**, and **Bala** would be a left branch from **Barmouth**.



A **binary tree** is a particularly suitable data structure if **very fast searching** is important. However, frequently adding and deleting records can cause problems. When a node is removed the whole tree has to be reconstructed. Therefore, if **data is constantly changing**, a **linked list** may be a more suitable data structure to use.

As in the case of a linked list, a **binary tree** can be represented using **arrays**. Two pointer arrays will be required, along with a **start pointer** to indicate the position of the **root node**.

	<b>Town</b>	<b>Left pointer</b>	<b>Right Pointer</b>
[0]	Dolgellau	1	2
[1]	Barmouth	5	3
[2]	Porthmadog	-1	4
[3]	Corris	-1	-1
[4]	Tywyn	-1	-1
[5]	Bala	-1	-1
[6]			

In this chapter we will develop a binary tree system for the tourist information centre, and investigate how records can be added, displayed and output.

Begin the project in the standard way. Close all previous projects, then set up a **New Project**. Give this the name **binaryTree**, and ensure that the **Create Main Class** option is not selected.

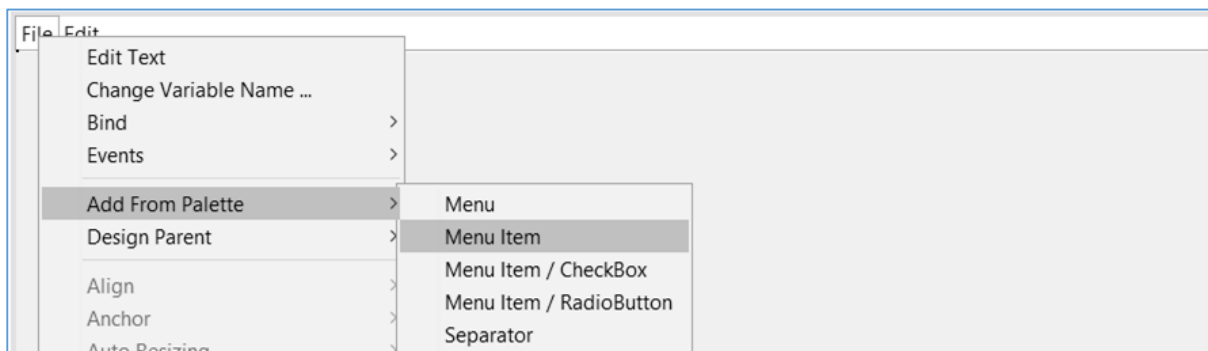
Return to the NetBeans editing page. Right-click on the **binaryTree** project, and select **New / JFrame Form**. Give the **Class Name** as **binaryTree**, and the **Package** as **binaryTreePackage**:

Return to the NetBeans editing screen.

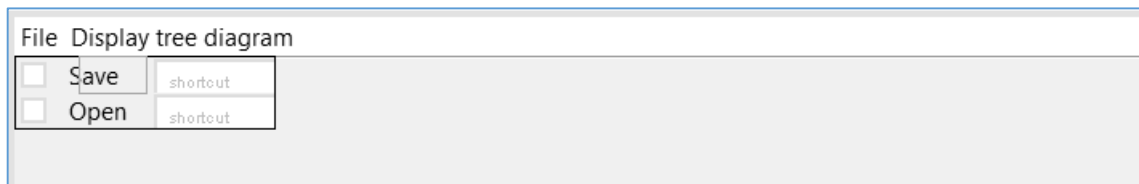
- Right-click on the **form**, and select **Set layout / Absolute layout**.
- Go to the **Properties** window on the bottom right of the screen and click the **Code** tab. Select the option: **Form Size Policy / Generate pack() / Generate Resize code**.
- Click the Source tab above the design window to open the program code. Locate the main method. Use the + icon to open the program lines and change the parameter "**Nimbus**" to "**Windows**".

Run the program and accept the **main** class which is offered. Check that a blank window appears and has the correct size and colour scheme. Close the program and return to the editing screen. Click the Design tab to move to the form layout view.

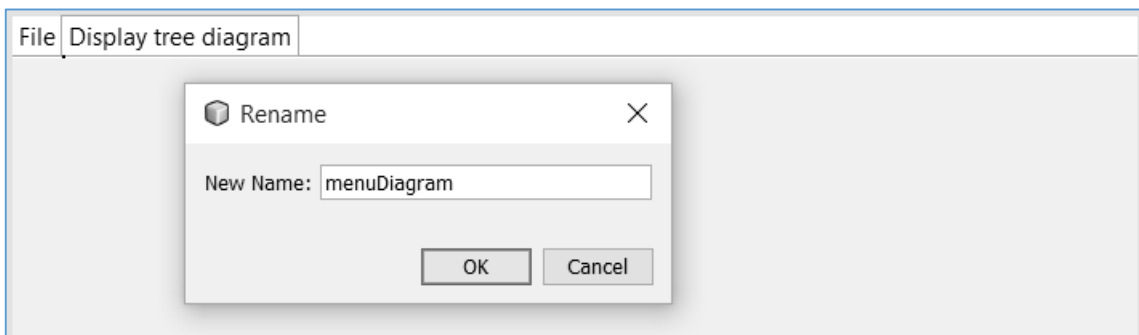
Select a **Menu Bar** component from the **Palette** and drag this onto the form. Right-click on the **File** menu option and select **Add From Palette / Menu Item**. Right-click again on the **File** menu option and add a second **Menu Item**.



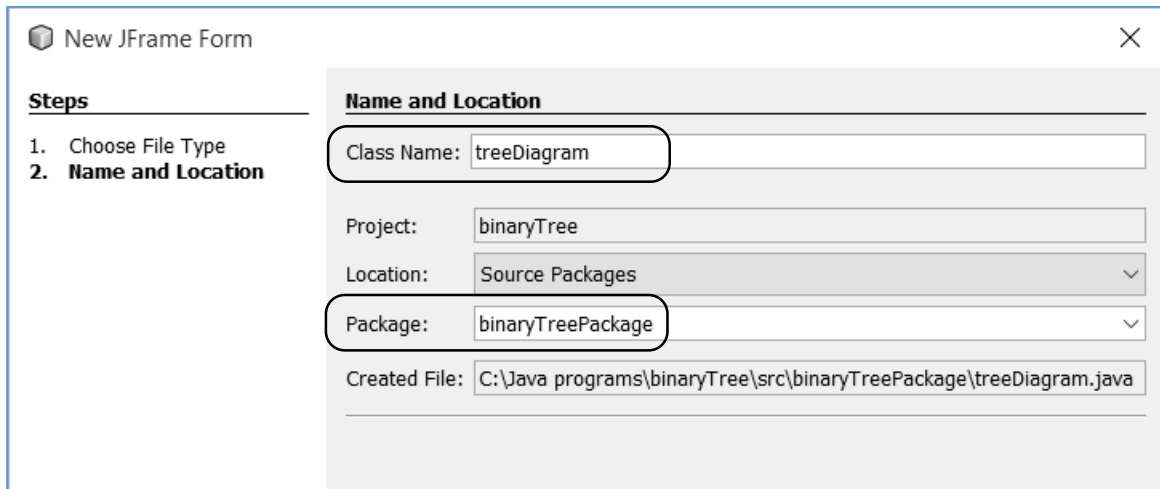
Right-click on each of the menu items, and edit the text values to '**Save**' and '**Open**'. Change the text of the **Edit** menu option to '**Display tree diagram**'.



Right-click again on the menu options '**Save**', '**Open**' and '**Display tree diagram**'. Rename these as **menuSave**, **menuOpen** and **menuDiagram**.



Create another form by right-clicking on **binaryTreePackage** in the **Projects** window, then selecting **New / JFrame Form**. Give the Class Name as '**treeDiagram**'. Leave the Package name as '**binaryTreePackage**'.



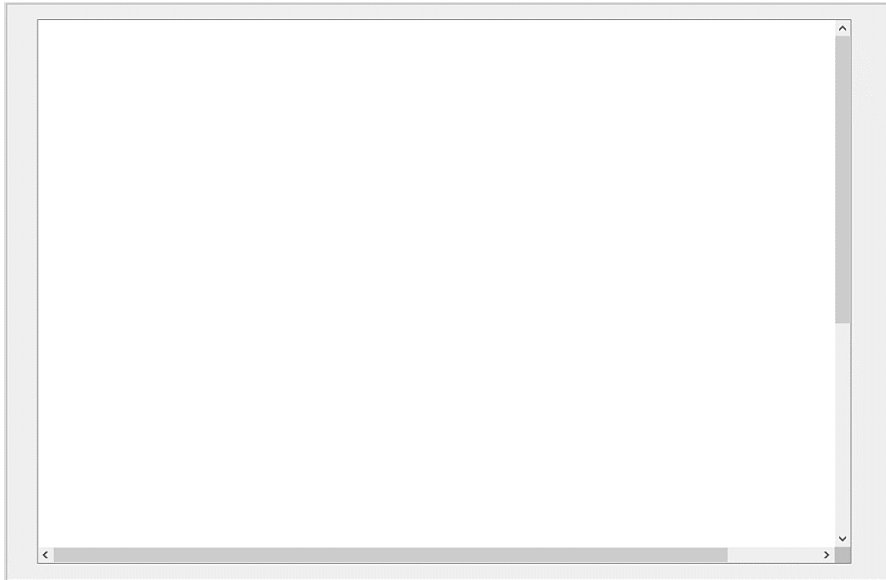
Return to the NetBeans editing screen.

Right-click on the treeDiagram.java **form**, and select **Set layout / Absolute layout**.

Go to the **Properties** window on the bottom right of the screen and click the **Code** tab. Select the option: **Form Size Policy / Generate pack() / Generate Resize code**.

Set the **defaultCloseOperation** property to **HIDE** by selecting from the drop down list.

Add a **Scroll Pane** component to the form. The diagram we will be producing is large, so make the form and scroll pane about 1000 pixels wide by 800 pixels high. Select a **Panel** component, then drag and drop this in the middle of the scroll pane. Rename the panel as **pnDiagram**. Set the **background** property to **White**, and the **preferredSize** property to **[1200,1200]**. Vertical and horizontal scroll bars should appear on the panel.



Use the tab at the top of the editing window to move to the **binaryTree.java** page. Click the Design tab to move to the form layout screen.

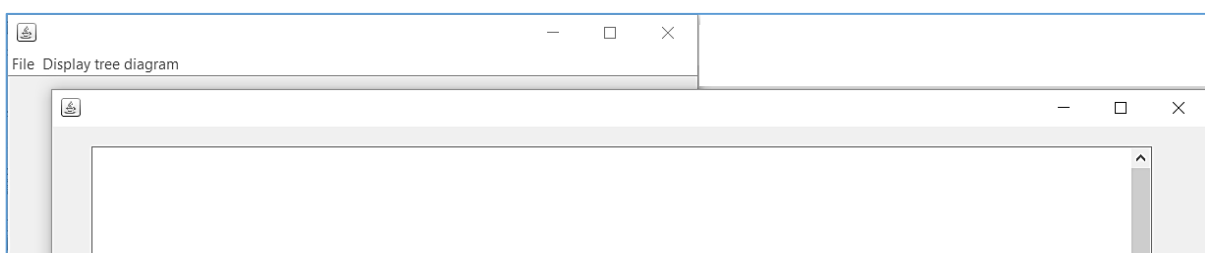
Select the '**Display tree diagram**' menu option. Go to the Properties window and select the **Events** tab. Scroll down to locate the **mouseClicked** event and accept **menuDiagramMouseClicked( )** from the drop down list.

menuSelected	<none>	▼	...
mouseClicked	menuDiagramMouseClicked	▼	...
mouseDragged	menuDiagramMouseClicked	▼	...
mouseEntered	<none>	▼	...

A menu click method will be created. Add a line of code to open the **treeDiagram.java** page.

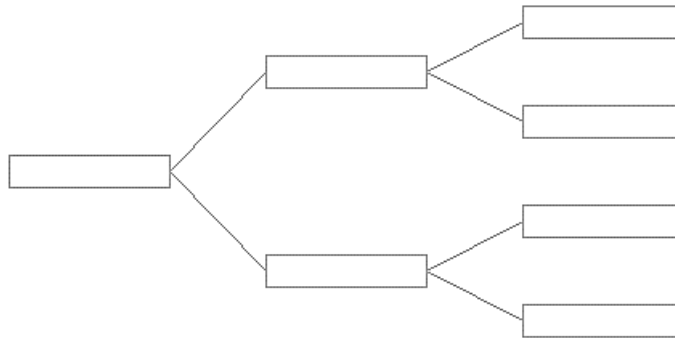
```
private void menuDiagramMouseClicked(java.awt.event.MouseEvent evt) {
    new treeDiagram().setVisible(true);
}
```

Run the program. Click the '**Display tree diagram**' menu option and check that the **treeDiagram** form opens. Clicking the cross in the corner should close the treeDiagram form, but leave the main program window running.



Close the program window. Use the tab at the top of the NetBeans editing screen to move to the **treeDiagram.java** page.

We will begin the program by producing an empty outline diagram of the binary tree. When data items are entered, they can then be displayed in the boxes of the diagram. It will be more convenient for the layout of the screen display if we draw the binary tree horizontally:



Click the **Source** tab to move to the program code display. Begin the programming by adding some Java modules which will be needed to produce the graphics:

```

package binaryTreePackage;

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics2D;

public class treeDiagram extends javax.swing.JFrame {

```

Use the **Design** tab to move back to the form layout view, and click on the white panel on the form. Select the **Events** tab at the top of the Properties window, then locate the **mouseMoved** event. Accept the **pnDiagramMouseMoved** from the drop down list.

mouseEntered	<none>
mouseExited	<none>
mouseMoved	pnDiagramMouseMoved
mousePressed	pnDiagramMouseMoved
mouseReleased	<none>
mouseWheelMoved	<none>

We will produce a method to draw boxes for the nodes of the binary tree. The **node()** method will use two input variables to specify the horizontal and vertical position where the box is to be drawn on the panel. Add a line of code to call the method to draw the root node.

```

private void pnDiagramMouseMoved(java.awt.event.MouseEvent evt) {

    node(40,500);

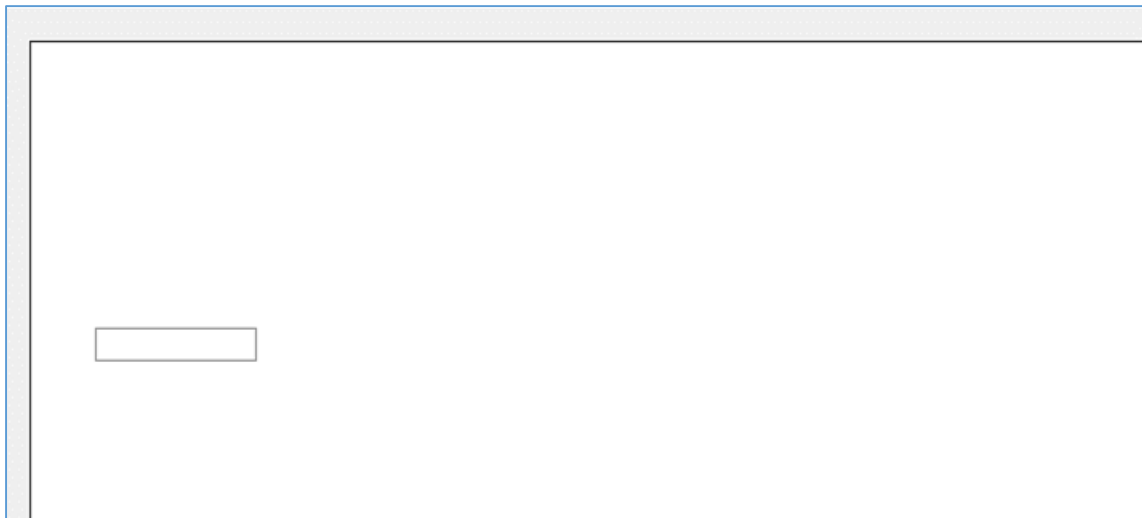
}

```

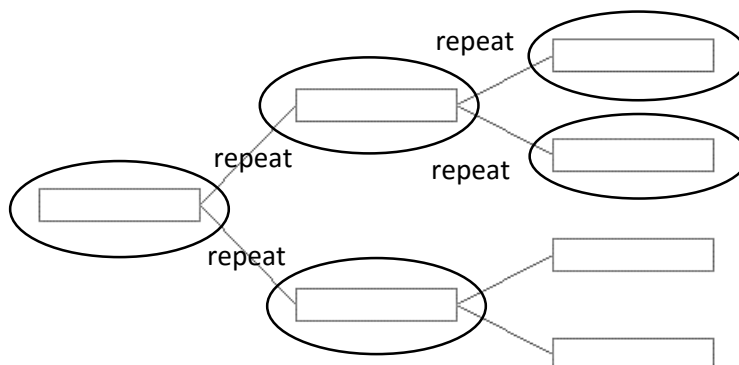
Immediately below the `MouseClicked` method, add the ***node()*** method.

```
private void node( int xpos, int ypos)
{
    Graphics2D g = (Graphics2D) pnlDiagram.getGraphics();
    int height=20;
    g.setColor(Color.lightGray);
    g.drawRect(xpos,ypos,100,height);
}
```

Run the program. Select the '***Display tree diagram***' menu option. Move the mouse onto the white panel and check that a box is drawn. It should be possible to move this up and down using the scroll bar.

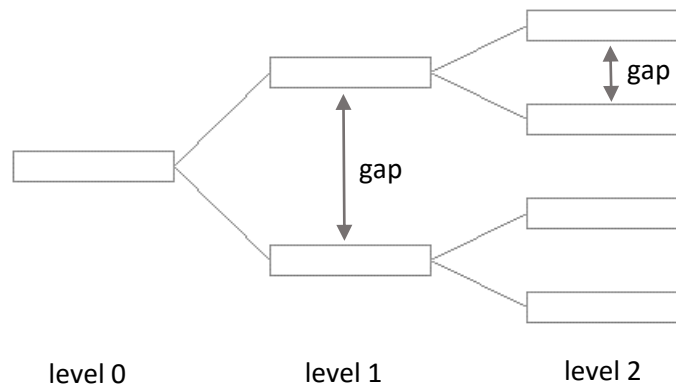


Close the program windows and return to the code editing page. We now need a strategy for drawing further nodes of the binary tree. You may notice that the same graphic pattern is being repeated ***inside itself*** as each new level of the diagram is produced. This suggests that a ***recursive*** drawing method can be developed.



At each recursive call, the x- and y-position of the box will change, so we need a way to transfer these coordinates during each call of the ***node()*** method.

As each recursive call opens, the **level** within the tree increases by one, and the **gap** between nodes is halved. These values will provide the parameters which we require to update the x- and y-coordinates.



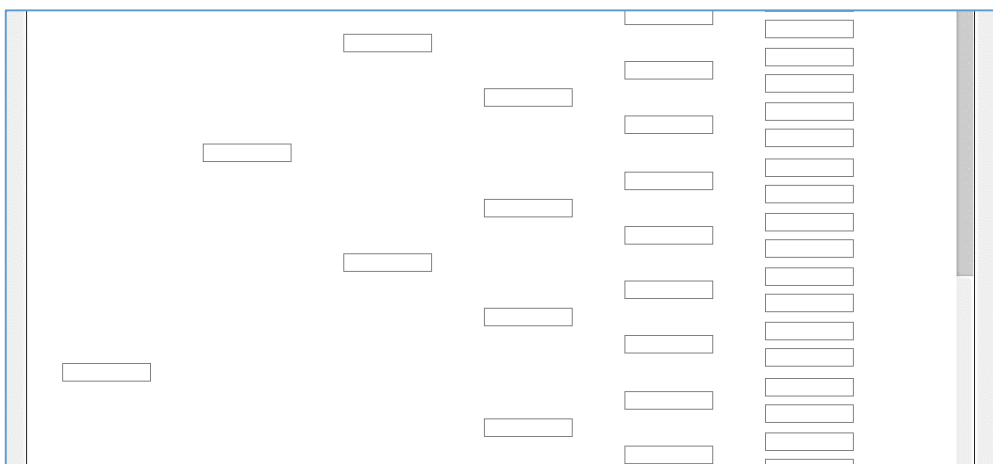
We will add **level** and **gap** as extra parameters for the **node()** method, and allow the method to call itself recursively up to level 4 of the tree structure. Edit or add the lines of code indicated below.

```
private void pnlDiagramMouseClicked(java.awt.event.MouseEvent evt) {
    node(0, 40,500,500);
}

private void node(int level, int xpos, int ypos, int gap)
{
    Graphics2D g = (Graphics2D) pnlDiagram.getGraphics();
    int height=20;
    g.setColor(Color.lightGray);
    g.drawRect(xpos,ypos,100,height);

    if (level<5)
    {
        xpos = xpos+160;
        node(level+1, xpos,ypos-gap/2,gap/2);
        node(level+1, xpos,ypos+gap/2,gap/2);
    }
}
```

Run the program. Select the '**Display tree diagram**' menu option. Move the mouse onto the white panel. Nodes should now appear for the series of levels of the binary tree.



Close the program windows and return to the code editing page. We just need to add the connecting lines to the diagram. This is slightly tricky, as we have both upwards and downwards branches, so the lines slant in different directions. A simple solution is to pass the x- and y-positions of the previous node as extra parameters of the `node()` method. Each recursive call can then draw a line from the current position to the end of the previous box. Edit or add the lines of code indicated below.

```
private void pnlDiagramMouseClicked(java.awt.event.MouseEvent evt) {
    node(0, 40,500,500,40,510);
}

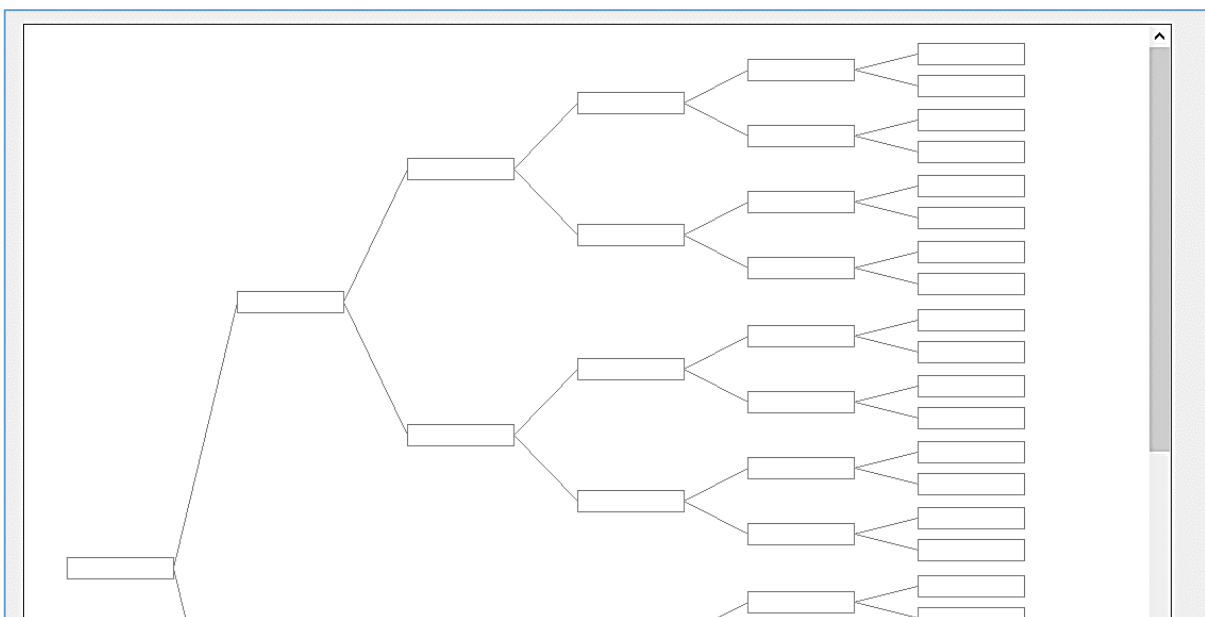
private void node(int level,int xpos,int ypos,int gap,int previousX,int previousY)
{
    Graphics2D g = (Graphics2D) pnlDiagram.getGraphics();
    int height=20;
    g.setColor(Color.lightGray);
    g.drawRect(xpos,ypos,100,height);

    g.drawLine(previousX,previousY,xpos,ypos+height/2);

    if (level<5)
    {
        xpos = xpos+160;

        node(level+1, xpos,ypos-gap/2,gap/2,xpos-60,ypos+10);
        node(level+1, xpos,ypos+gap/2,gap/2,xpos-60,ypos+10);
    }
}
```

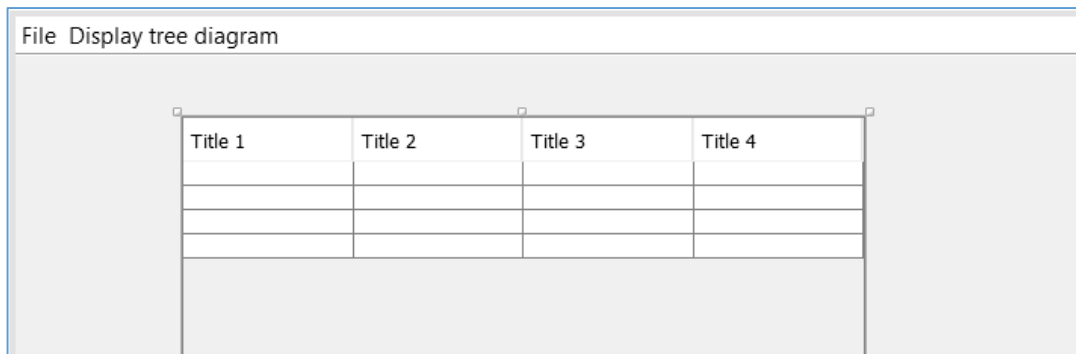
Run the program. Select the '**Display tree diagram**' menu option. Move the mouse onto the white panel. The nodes should now be linked to show the branching structure of the binary tree.





Close the program windows and return to the NetBeans editing screen. Use the tab to open the *binaryTree.java* page, then select the **Design** tab to move to the form layout view.

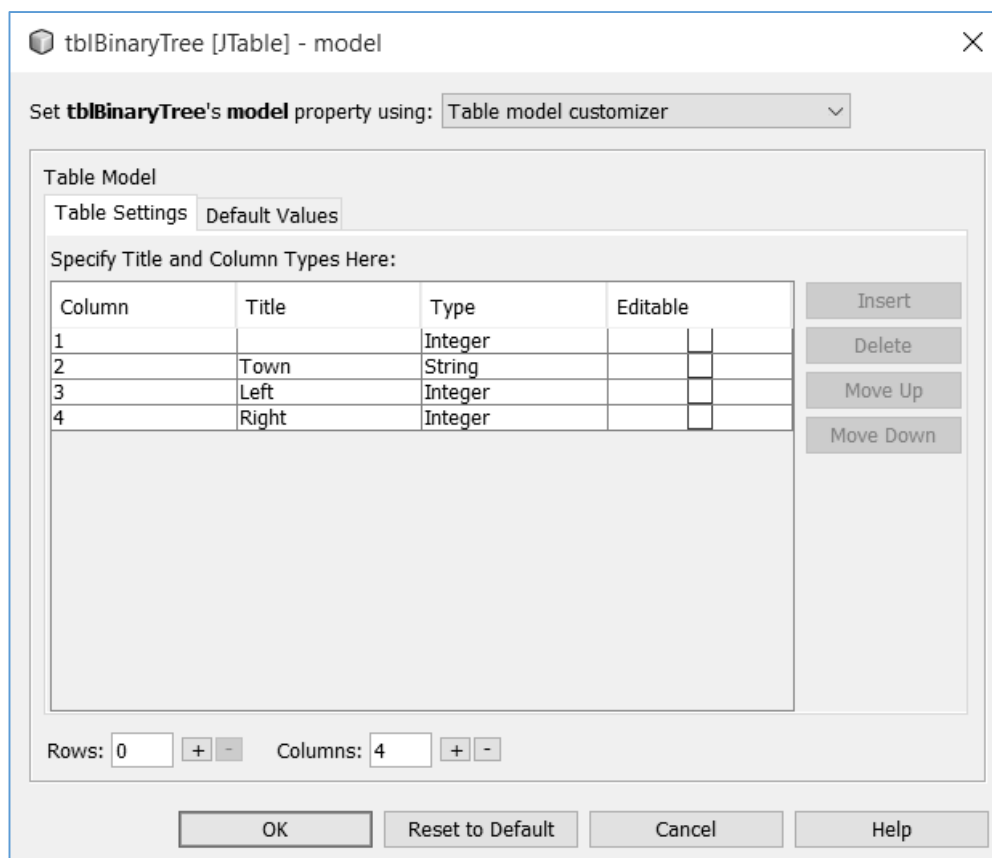
Add a Table component to the form and rename this as *tblBinaryTree*.



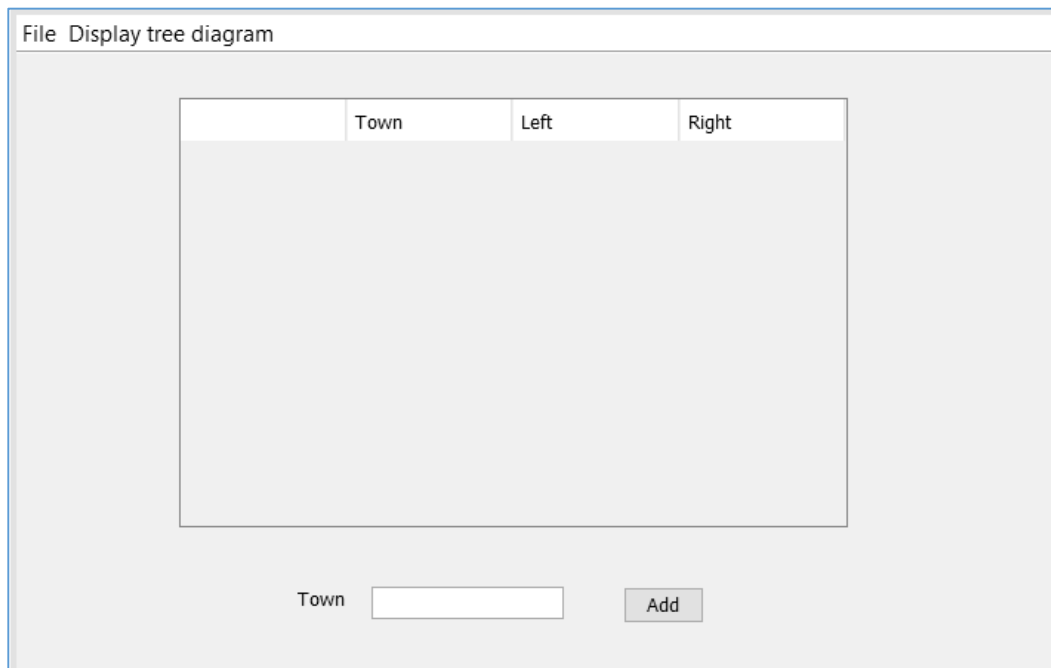
Go to the **Properties** window and locate the **model** property. Click in the right hand column to open the table editor. Set the number of rows to 0, and columns to 4. Set the Titles and data Types for the columns:

<no title>	<b>Integer</b>
<b>Town</b>	<b>String</b>
<b>Left</b>	<b>Integer</b>
<b>Right</b>	<b>Integer</b>

Remove the **Editable** ticks from each of the boxes.



Click the **OK** button to return to the form view and check that the table headings are displayed correctly. Below the table, add a label '**Town**' with a text field and button alongside. Rename the text field as **txtTown**. Edit the button caption to '**Add**' and the rename the button as **btnAdd**.



Use the **Source** tab to change to the program code view. We will begin by adding the Java modules needed for file handling and for the table display. We will then create arrays to store the names of the **towns**, and the **left** and **right pointers** for the binary tree, and set the start pointer '**root**' to have a value of 0.

```
package binaryTreePackage;

import java.io.File;
import java.io.IOException;
import java.io.RandomAccessFile;
import javax.swing.JOptionPane;
import javax.swing.table.DefaultTableModel;

public class binaryTree extends javax.swing.JFrame {

    String[] townData=new String[50];
    int[] left = new int[50];
    int[] right=new int[50];
    int root=0;
    static String filename = "binaryTree.dat";

    public binaryTree() {
        initComponents();
    }
}
```

Click the **Design** tab to return to the form layout view, then double click the '**Add**' button to produce a button click method.

Insert a line of code in the button click method to call **add()**, then begin the **add()** method immediately below.

```
private void btnAddActionPerformed(java.awt.event.ActionEvent evt) {
    add();
}

private void add()
{
}
```

We will work first on the code needed to insert a data item into the root node of an empty tree. Add lines of code to the **add()** method to do this.

```
private void add()
{
    String town=txtTown.getText();
    int position=0;
    Boolean finished;
    Boolean leftBranch=false;
    String currentData;

    if(townData[root]==null)
    {
        townData[root]=town;
        left[root]=-1;
        right[root]=-1;
    }
    txtTown.setText("");
    displayTable();
}
```

We begin by collecting the town name from the **txtTown** text field, then we define variables which will be needed by the method. The next block of code checks whether the root node is currently empty; if so, the town name is stored at this point and the left and right pointers initialised to -1 values. We finally clear the text field and call a method to display the updated array values in the table.

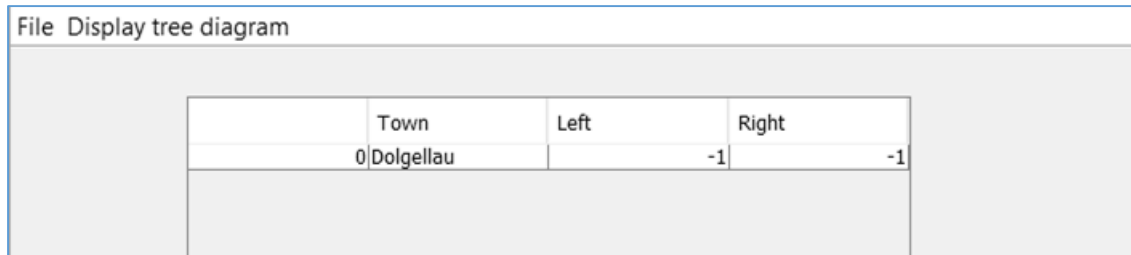
Insert the **displayTable()** method immediately below the **add()** method, as shown on the next page. This uses a loop to check each of the elements of the **townData[]** array. If an entry is found, the name of the town, along with the left and right pointer values, are added as a new line of the table.

```

private void displayTable()
{
    DefaultTableModel model = (DefaultTableModel)tblBinaryTree.getModel();
    model.setRowCount(0);
    for (int i=0; i<50;i++)
    {
        if (townData[i]!=null)
        {
            Object[] row = { i, townData[i],left[i],right[i]};
            model.addRow(row);
        }
    }
}

```

Run the program. Enter '**Dolgellau**' in the text box, then click the '**Add**' button. Check that the town name and the pointer values are correctly displayed in the table.



	Town	Left	Right
0	Dolgellau	-1	-1

Close the program window, return to the code editing screen and locate the **add()** method.

If another town name is entered when the tree already contains data, we must search the tree to find the position at the end of a branch where the new data item should be attached. An algorithm for this procedure is:

```

start at the root node
LOOP while the position to add the new data has not yet been found
    get the data item at the current node
    IF the new data item comes alphabetically before this item THEN
        IF the left pointer at this node has a value of -1 THEN
            we have found the position to add the new record
        ELSE
            set the new current position to this left pointer value
        ENDIF
    ELSE (then new data item comes alphabetically after this item)
        IF the right pointer at this node has a value of -1 THEN
            we have found the position to add the new record
        ELSE
            set the new current position to this right pointer value
        ENDIF
    ENDIF
END LOOP

```

Add the lines of code to locate the position where the new data item should be added to the binary tree.

```
if(townData[root]==null)
{
    townData[root]=town;
    left[root]=-1;
    right[root]=-1;
}

else
{
    finished=false;
    while(finished==false)
    {
        currentData=townData[position];
        if (town.compareTo(currentData)<0)
        {
            leftBranch=true;
            if (left[position]<0)
            {
                finished=true;
            }
            else
            {
                position=left[position];
            }
        }
        else
        {
            leftBranch=false;
            if (right[position]<0)
            {
                finished=true;
            }
            else
            {
                position=right[position];
            }
        }
    }
}

}

}

txtTown.setText("");
displayTable();
```

Once the position to add the new record has been found, several tasks must be carried out:

- The next available storage location must be found in the arrays.
- The new town name must be stored at this location, along with left and right pointer values of -1 to indicate the end of the branch.
- Either the left or right pointer from the previous node must be set to the location of the new record.

Add lines of code to carry out these tasks:

```

else
{
    leftBranch=false;
    if (right[position]<0)
    {
        finished=true;
    }
    else
    {
        position=right[position];
    }
}
}

int i=0;
while(townData[i]!=null)
{
    i++;
}
townData[i]=town;
left[i]=-1;
right[i]=-1;
if (leftBranch==true)
{
    left[position]=i;
}
else
{
    right[position]=i;
}
}
txtTown.setText("");
displayTable();

```

Run the program. Add a sequence of towns in the order: **Dolgellau, Barmouth, Corris, Porthmadog, Harlech, Tywyn, Bala, Blaenau**. Check that the pointer values are set correctly to create the binary tree.

	Town	Left	Right
0	Dolgellau	1	3
1	Barmouth	6	2
2	Corris	7	-1
3	Porthmadog	4	5
4	Harlech	-1	-1
5	Tywyn	-1	-1
6	Bala	-1	-1
7	Blaenau	-1	-1



Close the program window and return to the editing screen. Use the **Design** tab to move back to the form layout view, then double click the '**Open**' menu option to create a method. Add lines of code which will read in the start pointer value, followed by each of the town records. Finally, the **displayTable()** method is called to display the data.

```
private void menuOpenActionPerformed(java.awt.event.ActionEvent evt) {
    String tRecordID;
    String tData;
    String tLeft;
    String tRight;
    try
    {
        int position;
        RandomAccessFile file = new RandomAccessFile(filename, "r");
        byte[] bytes = new byte[2];
        file.read(bytes);
        String s=new String(bytes);
        s=s.trim();
        root=Integer.valueOf(s);
        int readingCount=(int) file.length()/39;
        for (int i=0; i<readingCount; i++)
        {
            position=i*39+5;
            file.seek(position);
            bytes = new byte[39];
            file.read(bytes);
            s=new String(bytes);
            tRecordID=s.substring(0,2).trim(); s=s.substring(2);
            tData=s.substring(0,30).trim(); s=s.substring(30);
            tLeft=s.substring(0,2).trim(); s=s.substring(2);
            tRight=s.substring(0,2).trim();
            int n=Integer.valueOf(tRecordID);
            townData[n]=tData;
            left[n]=Integer.valueOf(tLeft);
            right[n]=Integer.valueOf(tRight);
        }
        file.close();
        displayTable();
    }
    catch(IOException e)
    {
        JOptionPane.showMessageDialog(binaryTree.this, "File error");
    }
}
```

Run the program. Click the '**Open**' menu option and check that the previous data is loaded correctly. Add two additional towns: **Pwllheli** and **Aberystwyth**. Click the '**Save**' menu option, then close the program window.

Re-run the program and click the '**Open**' menu option. Check that the full list of towns including **Pwllheli** and **Aberystwyth** is loaded and displayed correctly, as shown below.



File Display tree diagram

	Town	Left	Right
0	Dolgellau	1	3
1	Barmouth	6	2
2	Corris	7	-1
3	Porthmadog	4	5
4	Harlech	-1	-1
5	Tywyn	8	-1
6	Bala	9	-1
7	Blaenau	-1	-1
8	Pwllheli	-1	-1
9	Aberystwyth	-1	-1

Close the program window and return to the NetBeans editing screen. We will now work on the procedure to display the data as a **binary tree diagram**. The array data will be needed by methods on the **treeDiagram.java** page, so we will transfer it by means of a **data class**.

Go to the **Projects** window at the top left of the screen and locate the **binaryTreePackage** folder. Right-click on **binaryTreePackage** and select **New / Java Class**. Give the Class Name as '**data**', leaving the Package name as '**binaryTreePackage**'. Click the **Finish** button.

New Java Class

**Steps**

1. Choose File Type
2. **Name and Location**

**Name and Location**

Class Name:

Project:

Location:

Package:

Created File:

The **data** class file will open. Add variables to hold the binary tree data items and start pointer value.

```
package binaryTreePackage;

public class data {

    public static int root;
    public static String[] townData=new String[50];
    public static int[] left = new int[50];
    public static int[] right=new int[50];

}
```

We will need to save the binary tree into the **data** class when the '**Display tree diagram**' option is selected from the Menu Bar of the main program.

Use the tab at the top of the editing window to move to the **binaryTree.java** page. Click the Source tab to move to the code editing screen, then scroll down the program listing to locate the **menuDiagramMouseClicked()** method. Add a line of code to call a **saveData()** method, then add this below **menuDiagramMouseClicked()**.

```
private void menuDiagramMouseClicked(java.awt.event.MouseEvent evt) {
    saveData();
    new treeDiagram().setVisible(true);
}

private void saveData()
{
    data.root = root;
    for (int i=0;i<50;i++)
    {
        data.townData[i]=townData[i];
        data.left[i] = left[i];
        data.right[i]=right[i];
    }
}
```

Return to the **treeDiagram.java** page and click the **Source** tab to move to the program code screen. We will begin by adding variables which will be needed to hold the data, and lines of code to set up a graphics colour and font.

```
package binaryTreePackage;

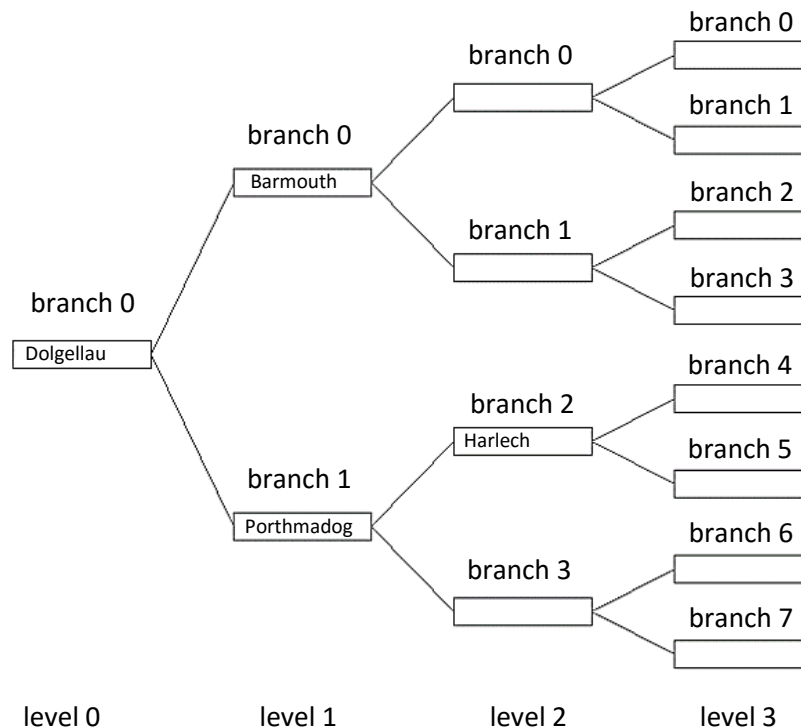
import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics2D;

public class treeDiagram extends javax.swing.JFrame {

    String[] townData=new String[50];
    int[] left = new int[50];
    int[] right=new int[50];
    int[] xNode=new int[50];
    int[] yNode=new int[50];
    int root;
    Font sanSerifFont = new Font("SanSerif", Font.PLAIN, 12);
    int red=0x99;
    int green=0xFF;
    int blue=0xFF;
    Color lightBlue = new Color(red,green,blue);

    public treeDiagram() {
        initComponents();
    }
}
```

In order to display the names of the towns in the correct positions on the tree diagram, we need to know the level and branch number where the name appears in the tree. For example: **Barmouth** should be drawn at **level 1, branch 0** and **Harlech** should be drawn at **level 2, branch 2**.



We need a way of finding the level and branch number for each town. Fortunately this can be done quite easily with a recursive method.

**Please note that the tree diagram in the program extends horizontally, rather than vertically.**

**For items earlier in the alphabet, branches move upwards rather than to the left.**

**For items later in the alphabet, branches move downwards rather than to the right.**

- We begin with the town at the **root node**. This has **level 0, branch 0**.
- If there is a **upwards branch**, the method can be called again recursively to find the level and branch number of the next node. The **level** will have **increased by 1**, and the number of the **branch** will have **doubled**. You can check this on the diagram above. Remember that the diagram has been rotated, so that a left branch is now shown as sloping upwards towards the next node.
- If there is a **downwards branch**, the method can again be called recursively to find the level and branch number of the next node. The **level** will again have **increased by 1**, but the number of the **branch** will have **doubled plus one added**. You can check this on the diagram, remembering that a right branch is now shown as sloping downwards towards the next node.

Locate the **treeDiagram()** method near the beginning of the program listing. We will add a call to a **getData()** method, as shown below, which will load the binary tree data. We will also call a recursive method **nodeOrder()** to find the level and branch number for each town.

```

public treeDiagram() {
    initComponents();

    getData();
    nodeOrder(root,0,0);
}

```

Add the **getData()** and **nodeOrder()** methods below **treeDiagram()**.

```

public treeDiagram() {
    initComponents();
    getData();
    nodeOrder(root,0,0);
}

private void getData()
{
    root=data.root;
    for (int i=0;i<50;i++)
    {
        townData[i]=data.townData[i];
        left[i] =data.left[i];
        right[i]=data.right[i];
    }
}

private void nodeOrder(int node, int level, int position)
{
    xNode[node]=level;
    yNode[node]=position;
    if (left[node]>0)
    {
        nodeOrder(left[node], level+1,position*2);
    }
    if (right[node]>0)
    {
        nodeOrder(right[node], level+1,position*2+1);
    }
}

```

As the level and branch for each town is found, these values are stored in the corresponding elements of the **xNode[]** and **yNode[]** arrays, so they are available when we draw the tree diagram.

Scroll down the program listing to find the **node()** method which you produced earlier. Edit the title line, as shown below, to include a **branch** parameter. (Please note that the title line should appear as a single line of code without a line break.) Add the variables **s** and **found** at the start of the method.

```

private void node(int level,int branch,int xpos,int ypos,int gap,
                  int previousX,int previousY)
{
    Graphics2D g = (Graphics2D) pnlDiagram.getGraphics();
    int height=20;

    String s="";
    Boolean found=false;

    g.setColor(Color.lightGray);
    g.drawRect(xpos,ypos,100,height);

```

Add or edit the lines of code shown below. As each node is being plotted, the computer uses a loop to check the arrays for a town at the corresponding level and branch number. If a town is found, the name will be displayed in a blue box. If no name is found at this node, a grey outline will be drawn instead. The method then calls itself recursively to process the next branches of the tree.

```

private void node(int level, int branch, int xpos, int ypos, int gap,
                  int previousX, int previousY)
{
    Graphics2D g = (Graphics2D) pnlDiagram.getGraphics();
    int height=20;
    String s="";
    Boolean found=false;

    for(int i=0; i<50; i++)
    {
        if (xNode[i]==level && yNode[i]==branch && townData[i]!=null )
        {
            s=townData[i];
            g.setColor(lightBlue);
            g.fillRect(xpos,ypos,100,height);
            g.setColor(Color.black);
            g.drawString(s,xpos+3,ypos+height/2+4);
            g.drawRect(xpos,ypos,100,height);
            g.drawLine(previousX,previousY,xpos,ypos+height/2);
            found=true;
        }
    }
    if (found==false)
    {
        g.setColor(Color.lightGray);
        g.drawRect(xpos,ypos,100,height);
        g.drawLine(previousX,previousY,xpos,ypos+height/2);
    }

    if (level<5)
    {
        xpos = xpos+160;

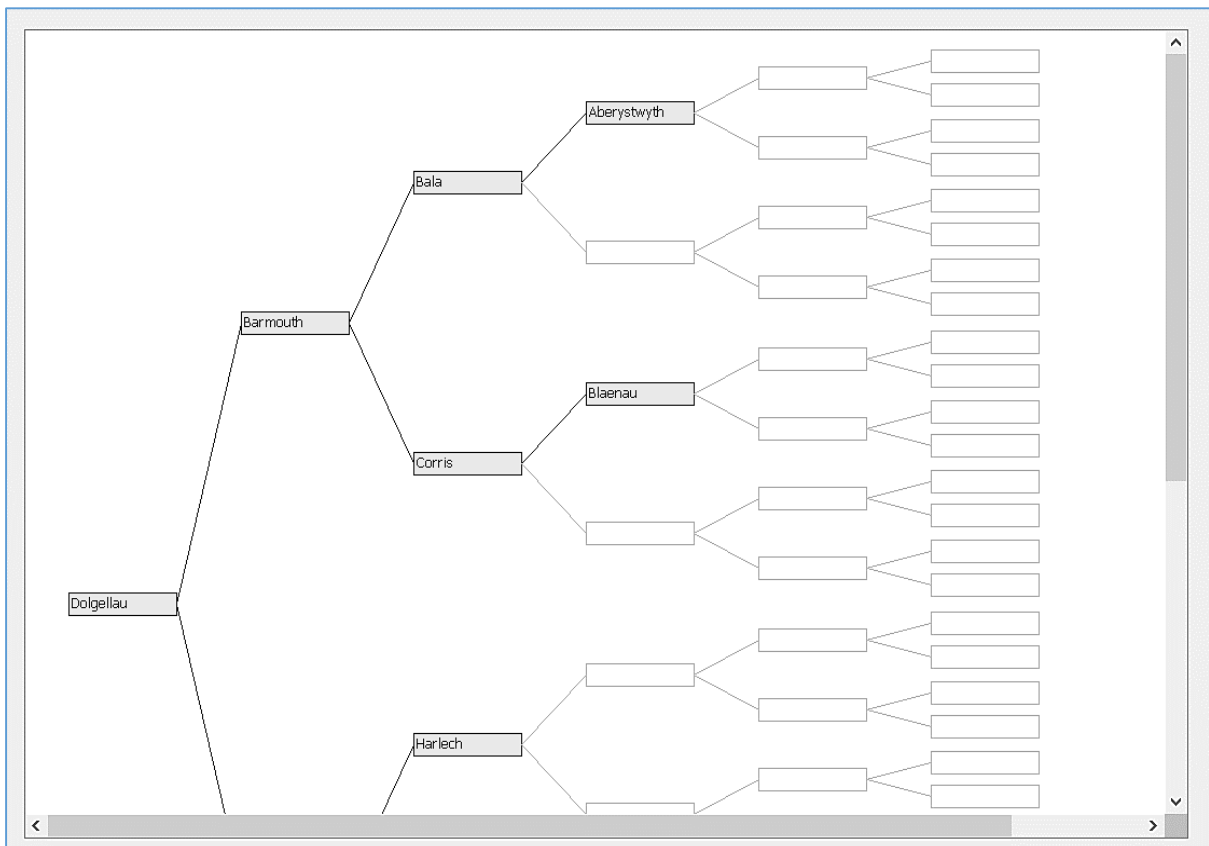
        node(level+1, branch*2, xpos,ypos-gap/2,gap/2,xpos-60,ypos+10);
        node(level+1, branch*2+1, xpos,ypos+gap/2,gap/2,xpos-60,ypos+10);
    }
}

```

We need to make one final change before running the program. Locate the **MouseMoved()** method and edit the call to **node()**, to include the extra parameter for the **branch** number.

```
private void pnlDiagramMouseMoved(java.awt.event.MouseEvent evt) {
    node(0,0,40,500,500,40,510);
}
```

Run the program. Click the '**Open**' menu option to load the data file, then click the '**Display tree diagram**' option. If all has gone well, the towns should now be displayed in their correct positions on the tree diagram.



Close the program windows and return to the NetBeans editing screen.

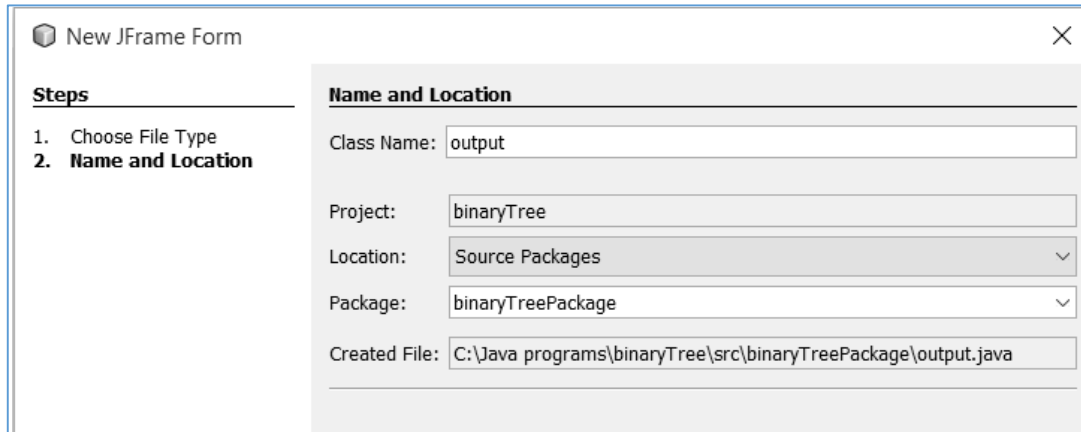
The main purpose of storing data in a binary tree is to allow **individual records** to be **retrieved quickly**. It is also quite easy, again using recursion, to print out **every data item** from the binary tree in **sorted order**. We will carry out these tasks next.

Click the tab to return to the **binaryTree.java** page, then the Design tab to select the form layout view. Select a **Menu** component from the Palette and add this to the **Menu Bar**, as shown below. Drag the Menu component onto the Menu Bar, being careful that the arrow of the mouse pointer is in the centre of the Menu Bar before releasing the mouse button.

Change the text to '**Search and output**', and rename the component as **menuOutput**.

File Display tree diagram Search and output

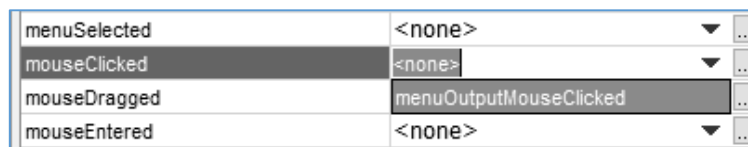
Go to the **Projects** window at the top left of the screen and locate the **binaryTreePackage** folder. Right click on **binaryTreePackage**, then select **New / JFrame Form**. Give the Class Name as **output**, leaving the Package name as **binaryTreePackage**.



Click **Finish** to return to the NetBeans editing screen.

- Right-click on the **output.java** form, and select **Set layout / Absolute layout**.
- Go to the **Properties** window on the bottom right of the screen and click the **Code** tab. Select the option: **Form Size Policy / Generate pack() / Generate Resize code**.
- Set the **defaultCloseOperation** property to **HIDE** by selecting from the drop down list.

Return to the **binaryTree.java** page and select the '**Search and output**' menu option. Go to the Properties window and select the **Events** tab. Locate the **mouseClicked** event, and accept **menuOutputMouseClicked** from the drop down list.



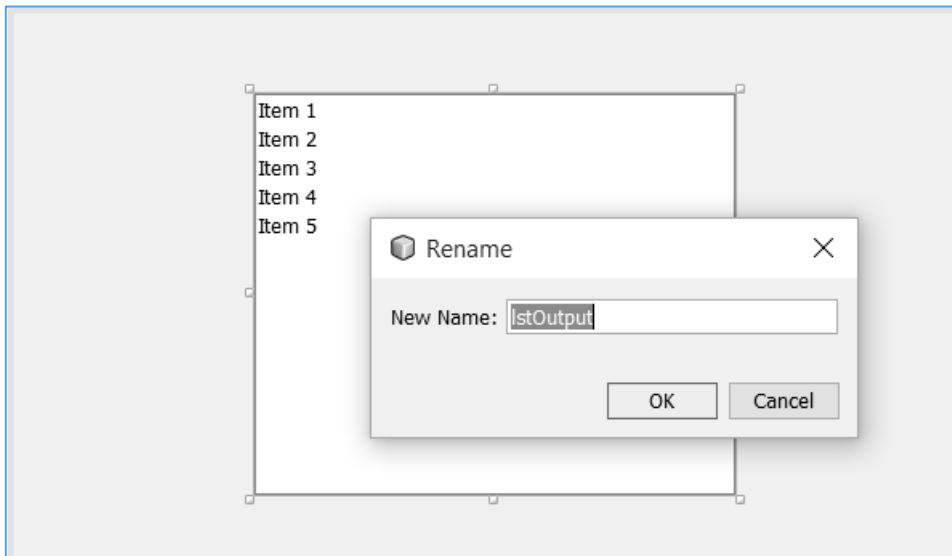
Add lines of code to the button click method to save the binary tree data, then open the **output.java** form. It is necessary to save the data so that it will be available when the new form opens.

```
private void menuOutputMouseClicked(java.awt.event.MouseEvent evt) {
    saveData();
    new output().setVisible(true);
}
```

Run the program. Click the '**Search and output**' menu option, and check that the **output.java** window opens correctly, and can be closed without exiting from the program.

Close the program windows and return to the NetBeans screen. Use the tab to select the **output.java** page.

Add a **List** component to the form, and rename this as **lstOutput**.



Use the **Source** tab to select the program code screen. Add a Java module needed by the **List** component, and define variables to hold the binary tree data. We then call the **getData()** method to load the binary tree data from the **data** class.

```

package binaryTreePackage;

import javax.swing.DefaultListModel;

public class output extends javax.swing.JFrame {

    int root;
    String[] townData=new String[50];
    int[] left = new int[50];
    int[] right=new int[50];
    DefaultListModel listModel = new DefaultListModel();

    public output() {
        initComponents();

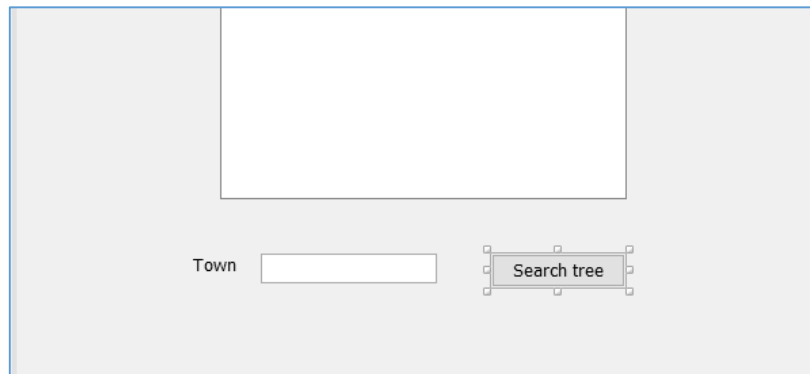
        getData();
        listModel.clear();
        lstOutput.setModel(listModel);
    }

    private void getData()
    {
        root=data.root;
        for (int i=0;i<50;i++)
        {
            townData[i]=data.townData[i];
            left[i] =data.left[i];
            right[i]=data.right[i];
        }
    }
}

```



Use the **Design** tab to return to the form layout view. Add a Label '**Town**' below the List box. Place a Text field alongside and rename this as **txtTown**. Finally, add a button with the caption '**Search tree**'. Rename the button as **btnSearch**.



Double click the button to create a method. Add the line of code to call a **search()** method, then begin the **search()** method immediately underneath. Insert the lines of code which initialise variables and collect the name of the required town from the **txtTown** text field.

```
private void btnSearchActionPerformed(java.awt.event.ActionEvent evt) {
    search();
}

private void search()
{
    int location= -1;
    String searchItem=txtTown.getText();
    int node=root;
    Boolean finished=false;
    listModel.clear();
}
```

The loop structure to search for a town is similar to the method we used previously to add a record to the tree.

```
start at the root node
LOOP while the required town has not been found AND there are still nodes to search
    get the data item at the current node
    IF the required town has not been found THEN
        IF the required town comes alphabetically before this item THEN
            branch upwards on the diagram if a left branch is present
        ELSE
            branch downwards on the diagram if a right branch is present
        ENDIF
    ENDIF
END LOOP
```

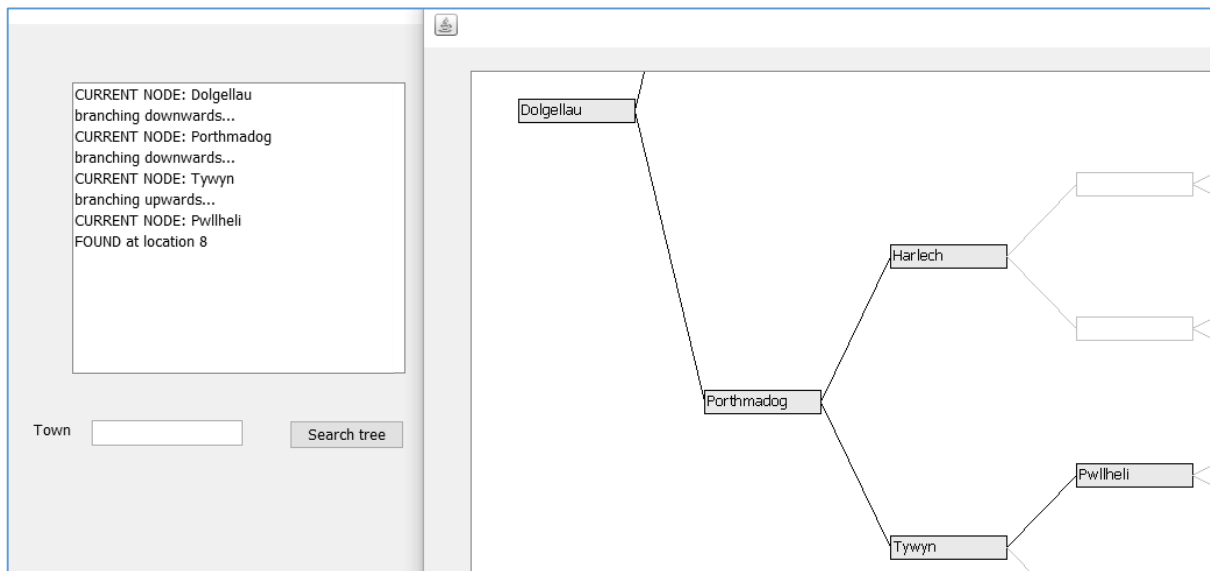
Add the lines of code to implement the search algorithm.

```
private void search()
{
    int location= -1;
    String searchItem=txtTown.getText();
    int node=root;
    Boolean finished=false;
    listModel.clear();

    while (finished==false)
    {
        listModel.addElement("CURRENT NODE: "+townData[node]);
        if(townData[node].compareTo(searchItem)==0)
        {
            finished=true;
            location=node;
            listModel.addElement("FOUND at location "+location);
        }
        else
        {
            if(townData[node].compareTo(searchItem)>0)
            {
                node=left[node];
                listModel.addElement("branching upwards...");
            }
            else
            {
                node=right[node];
                listModel.addElement("branching downwards...");
            }
            if(node==-1)
            {
                finished=true;
            }
        }
    }
    if (location<0)
    {
        listModel.addElement("NOT PRESENT in the tree");
    }
    lstOutput.setModel(listModel);
    txtTown.setText("");
}
}
```

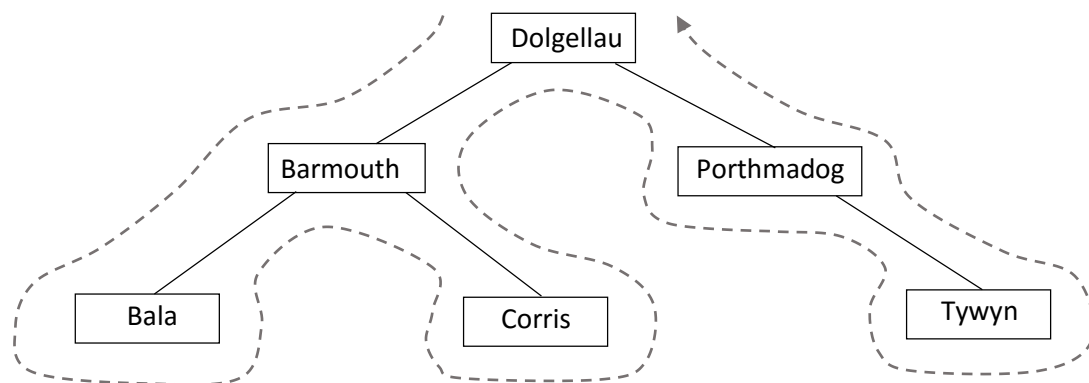
Run the program and load the data file. Click the menu option to open the *tree diagram window*, so that this can be used to check that the search is carried out correctly.

Click the menu option to open the *search window*. Enter the names of different towns, then check that the output displayed in the *List* corresponds with the correct search path on the tree diagram, as in the example below. Check also that a correct message is given if the town is not present in the tree.

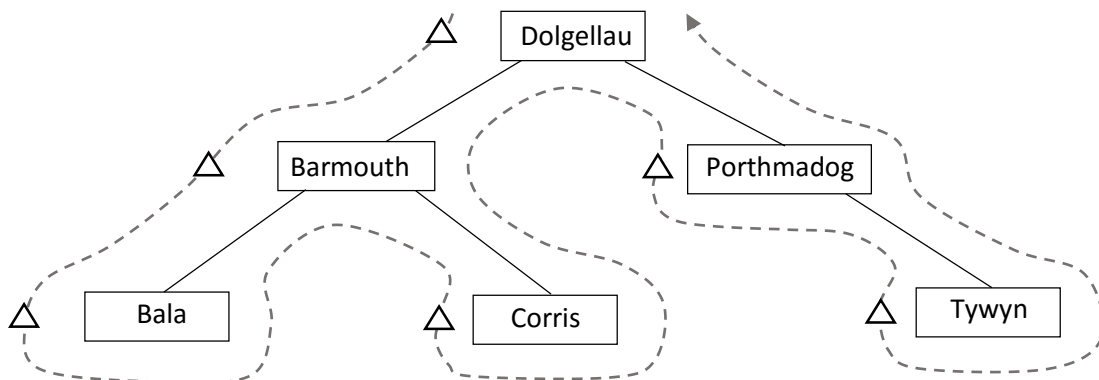


Close the program windows and return to the NetBeans editing screen.

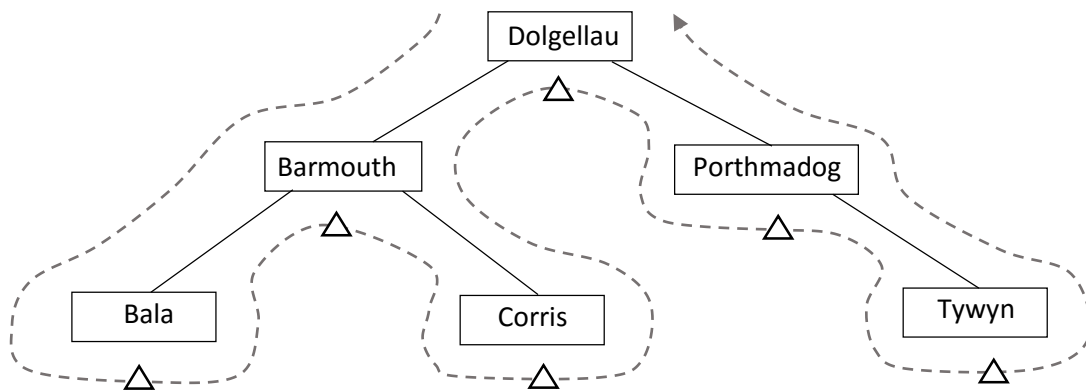
In addition to selecting a single record, it is possible to output all records of the binary tree. We will go back to considering a tree diagram drawn horizontally. Three different sequences are possible, which are known as **pre-order**, **in-order** and **post-order**. Imagine a line being traced anticlockwise around the outside of the binary tree, starting beside the root node:



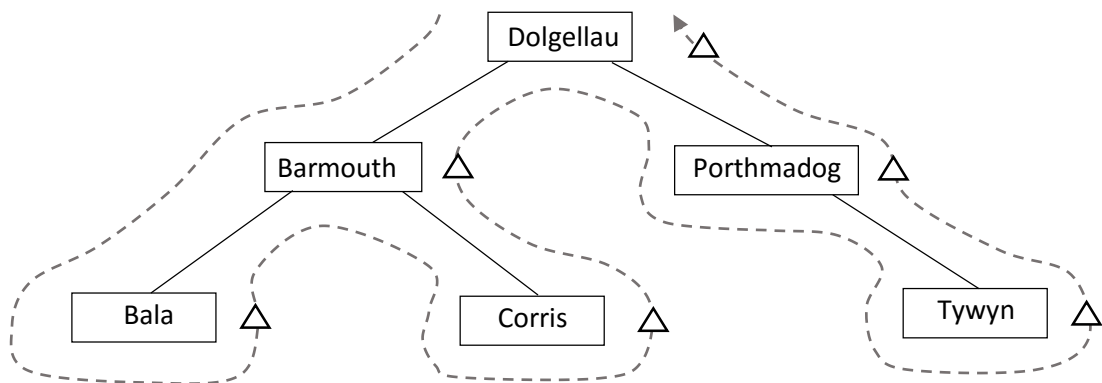
For a **pre-order** sequence, each town name is output as the line passes the **left** side of the node, giving the sequence: Dolgellau, Barmouth, Bala, Corris, Porthmadog, Tywyn.



For an *in-order* sequence, each town name is output as the line passes *beneath* the node, giving the sequence: Bala, Barmouth, Corris, Dolgellau, Porthmadog, Tywyn.



The *post-order* sequence outputs each town name as the line passes the *right* side of the node, giving: Bala, Corris, Barmouth, Tywyn, Porthmadog, Dolgellau.



The *in-order* sequence is probably the most commonly used, as this outputs the data in *alphabetical order*, but *pre-order* and *post-order* sequences are important in some software applications. All three of the sequences can be generated by recursive methods. We will now examine how output sequences can be programmed.

Use the *Design* tab to open the form layout view for the *output.java* page. Add a label 'Output tree' and three buttons with the captions '*pre-order*', '*in-order*' and '*post-order*'. Rename the buttons as *btnPreorder*, *btnInorder* and *btnPostorder*.

Town  Search tree

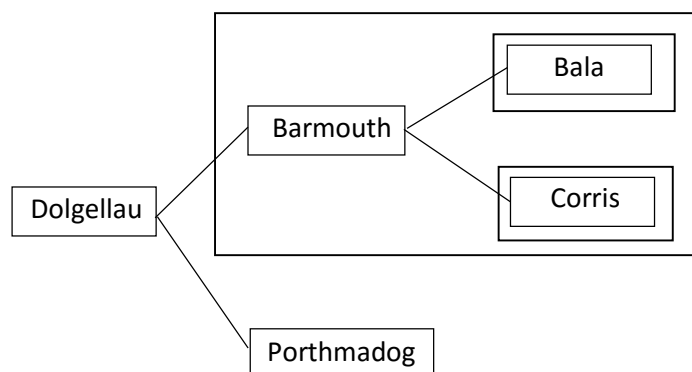
Output tree pre-order in-order post-order

Double click the pre-order button to create a method. Add lines of code to clear the List box, then call a recursive method *preOrder()*. Add this method immediately underneath.

```
private void btnPreoderActionPerformed(java.awt.event.ActionEvent evt) {
    listModel.clear();
    preOrder(root);
    lstOutput.setModel(listModel);
}

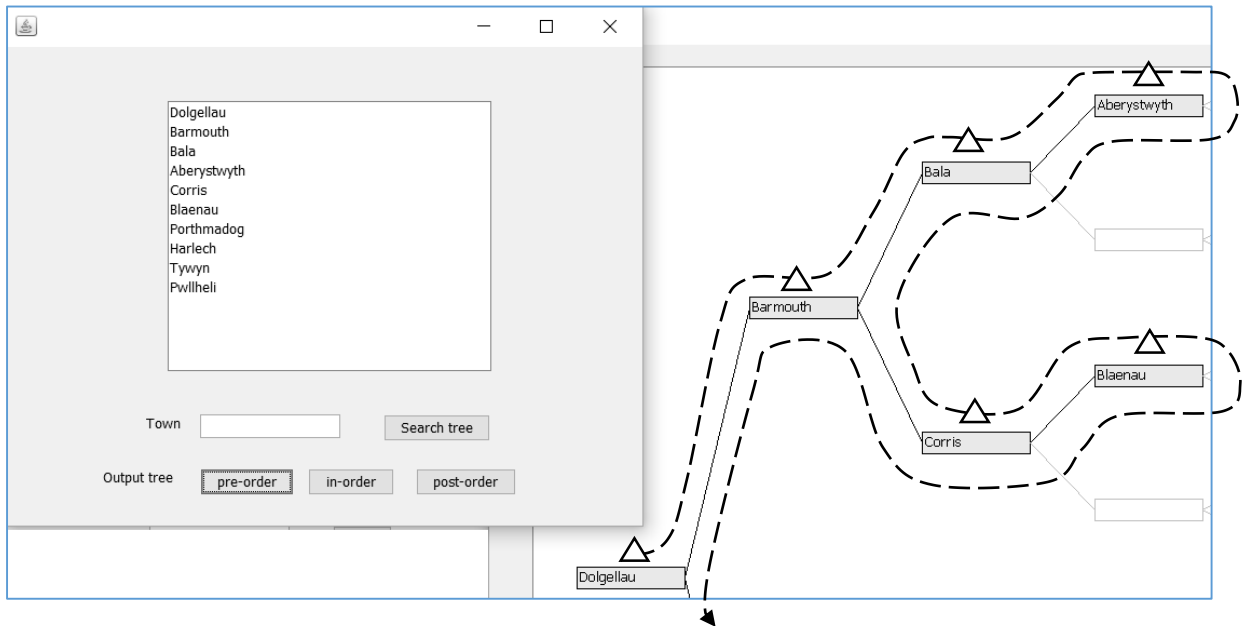
private void preOrder(int node)
{
    listModel.addElement(townData[node]);
    if (left[node]>0)
    {
        preOrder(left[node]);
    }
    if (right[node]>0)
    {
        preOrder(right[node]);
    }
}
```

*Pre-order()* begins by outputting the town at the root node, which in this example would be **Dolgellau**. The method then starts again recursively for the **sub-tree** with **Barmouth** as the **new root node**. After outputting Barmouth, the method starts yet again for the **sub-tree** with **Bala** as the **root node**. Bala is output. No further subtrees exist, so the recursive call to Bala closes and the method then attempts to process right subtrees from Barmouth. **Corris** is found and output next. Recursive calls close, returning to the root of the complete tree. Further recursive calls then open to reach **Porthmadog**.



Run the program. Load the data file, then open the **tree diagram** window so that the output can be verified against the tree structure.

Open the **Search and output** window, then click the **pre-order** button. Check that the output sequence of towns is correct. Remember that the tree diagram has been rotated, so the outline now runs clockwise around the nodes of the tree. The method begins by outputting the data at the current node, then checks for an **upward branch** to a sub-tree. Finally, it checks for a **downward branch**.



Close the program windows and return to the *output.java* page. Use the Design tab to move to the form layout view, then double click the in-order button to create a method. Add lines of code to clear the List, then call an *inOrder()* method. Add this method underneath.

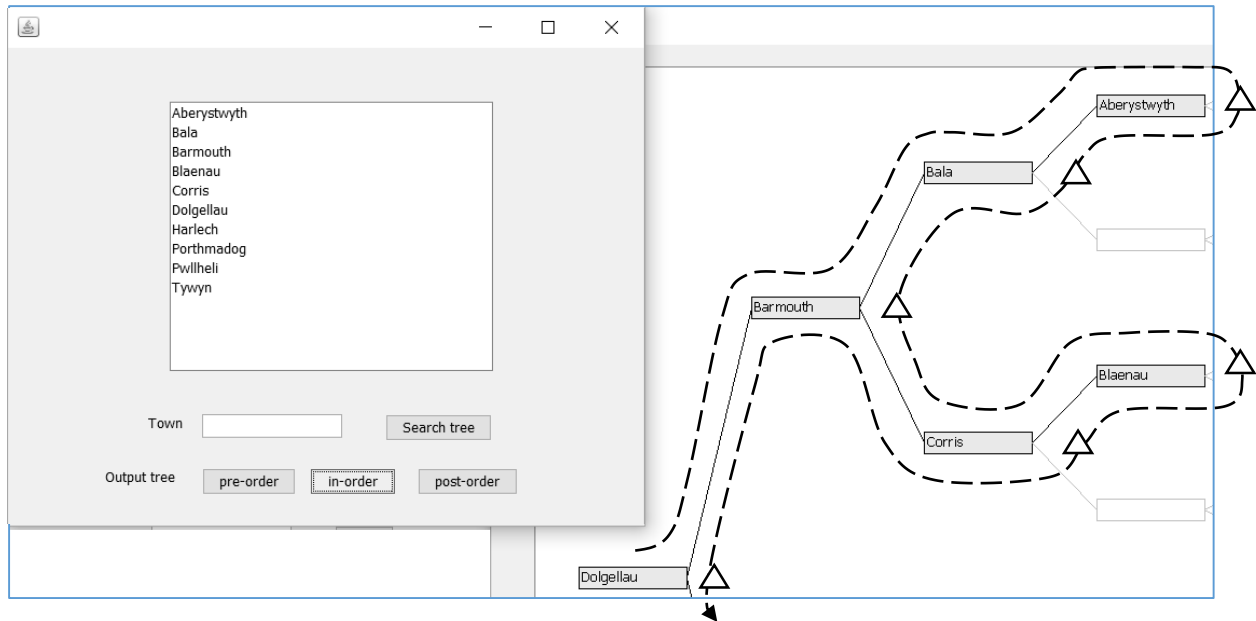
The *inOrder()* method is almost identical to the *preOrder()* method, except that recursive calls to any left subtrees are carried out before the data at the current node is output.

```
private void btnInorderActionPerformed(java.awt.event.ActionEvent evt) {
    listModel.clear();
    inOrder(root);
    lstOutput.setModel(listModel);
}

private void inOrder(int node)
{
    if (left[node]>0)
    {
        inOrder(left[node]);
    }
    listModel.addElement(townData[node]);
    if (right[node]>0)
    {
        inOrder(right[node]);
    }
}
```

Run the program. Load the data file, then open the tree diagram window so that the output can be verified against the tree structure.

Open the *Search and output* window, then click the *in-order* button. Check that the output sequence of towns is correct, as illustrated below.



Close the program windows and return to the *output.java* page. Use the *Design* tab to move to the form layout view, then double click the post-order button to create a method. Add lines of code to clear the List, then call an *postOrder()* method. Add this method underneath.

```
private void btnPostorderActionPerformed(java.awt.event.ActionEvent evt) {
    listModel.clear();
    postOrder(root);
    lstOutput.setModel(listModel);
}

private void postOrder(int node)
{
    if (left[node]>0)
    {
        postOrder(left[node]);
    }
    if (right[node]>0)
    {
        postOrder(right[node]);
    }
    listModel.addElement(townData[node]);
}
```

The *postOrder()* method is again very similar, except that recursive calls to both left and right subtrees are carried out before the data at the current node is output.

Run the program and again check that the output sequence is correct according to the tree diagram.