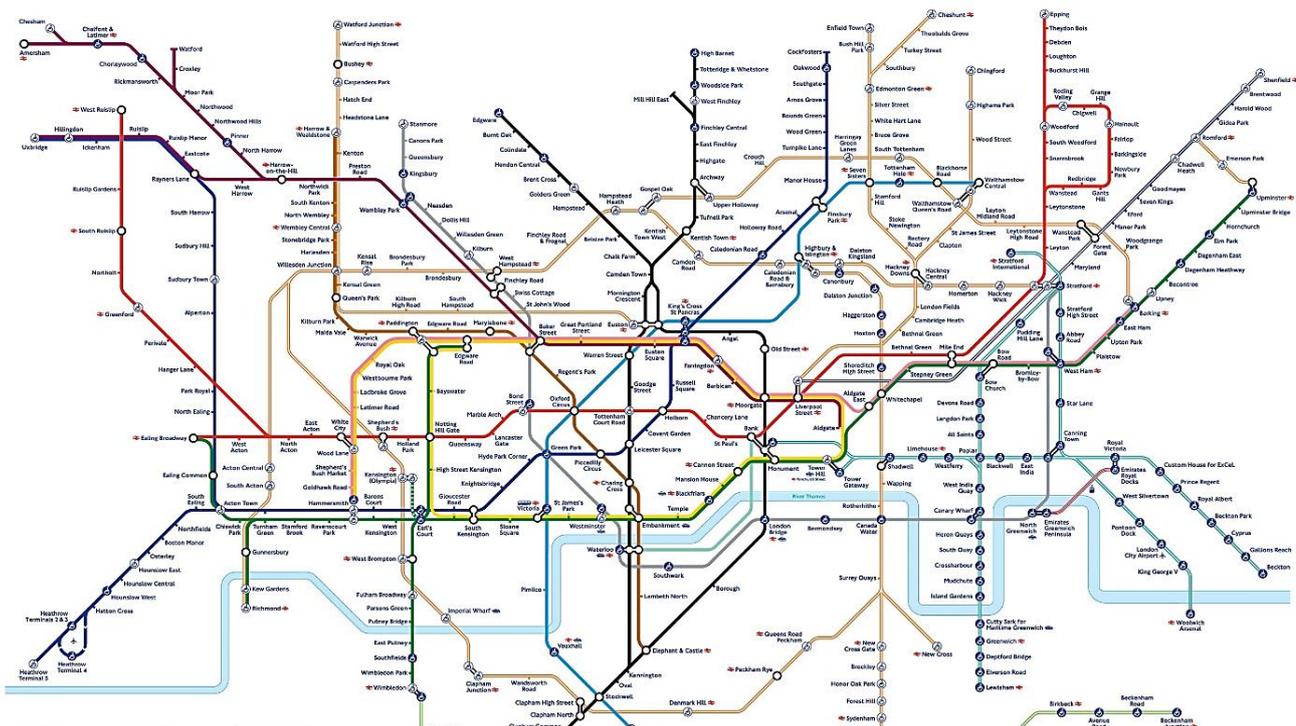


# London Underground

## Scenario

The objective of this project is to advise passengers on the best route to take between a starting point and destination on the London Underground.

The first London Underground line opened in 1863. Further lines have been progressively added up to the present day, resulting in a large and complex interconnected network. A diagrammatic map has been designed to help travellers navigate the system, with different underground lines identified by colour coding.



Due to the large number of interconnections, it is often possible to travel between two locations by a variety of routes. It may be necessary to change train and use a different underground line at one or more points during the journey. In this project, the search strategy will select routes which minimise the number of changes needed. It can be difficult to change platforms if stairs or escalators have to be negotiated, especially at busy times or when carrying luggage.

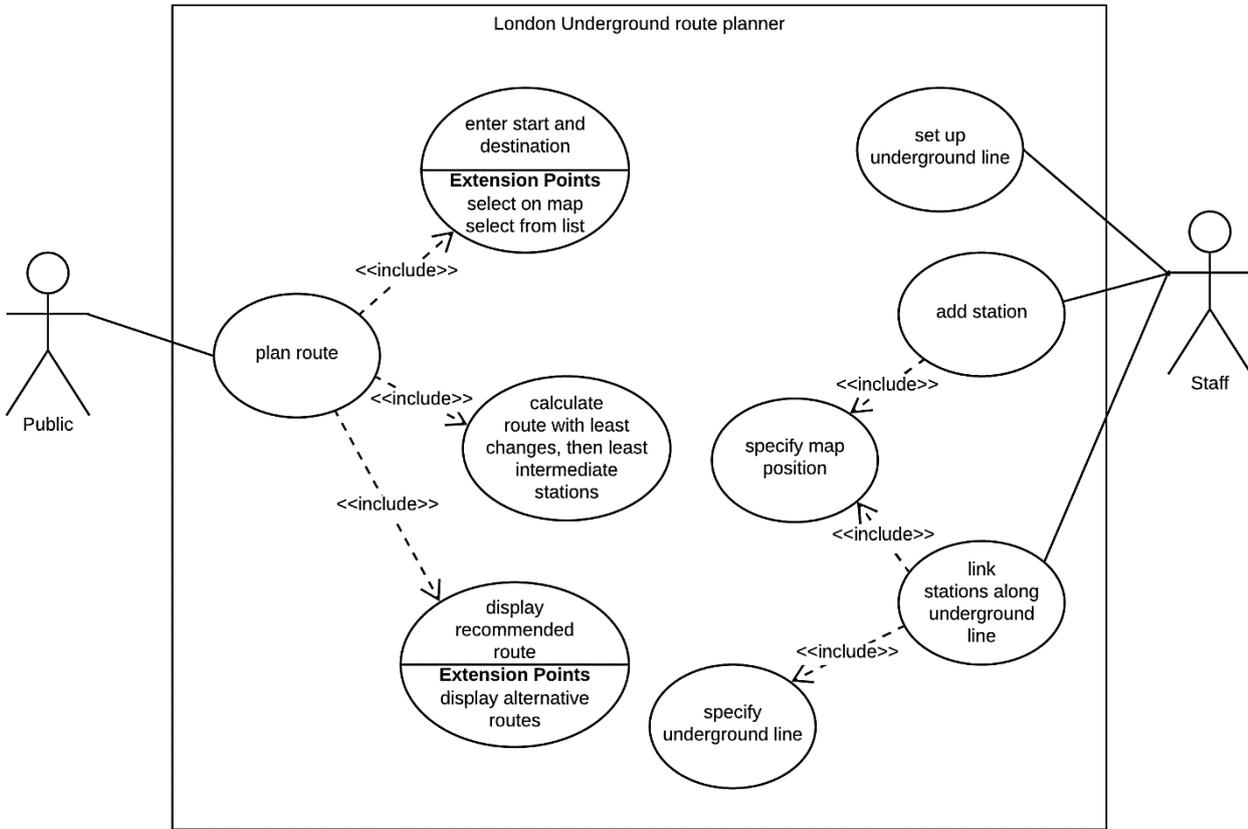
Once the minimum number of changes of train is found, there may still be several routes available. The program will attempt to find the quickest route. To simplify the problem, it is assumed that the journey times between each pair of stations is approximately equal. This is a reasonable assumption, at least for the central area of London where stations are closely spaced. If several routes are possible, the objective is to select the route passing through the minimum number of intermediate stations.

The recommended route will be presented to the user, but there should also be an option to see alternative routes. The user may have personal knowledge of the underground system and know how easy or difficult it is to change trains at a particular station, so may prefer one of the alternative routes.

In addition to the public function, the web site should allow staff to update the system in the event of new underground lines or stations opening, or existing stations closing.

**Design**

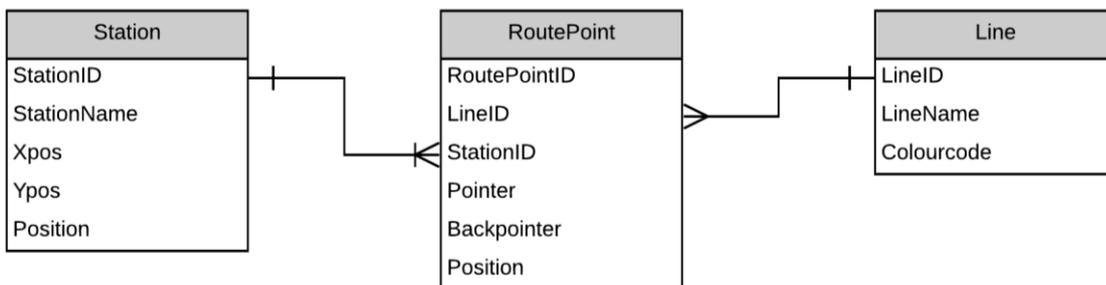
The requirements of the system are summarised in the use case diagram below.



Staff will use an on-screen display of the official underground map as a guide for adding stations and underground lines to the screen. The program can then construct a graphical display at run-time which resembles the published map as closely as possible.

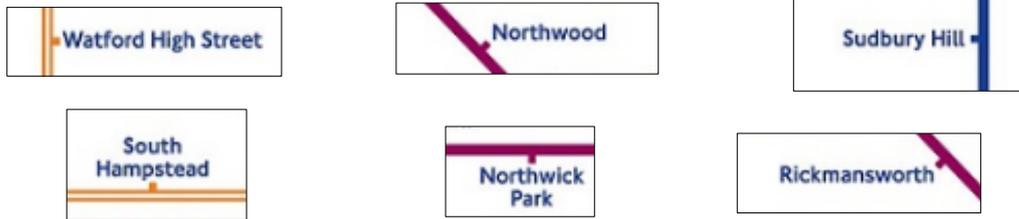
The user may select the start and destination by clicking on stations on the computer generated map, or may select from an alphabetical list of stations.

Attributes for underground lines and stations will be stored in database tables, and will be accessed by the map plotting and route planning algorithms.



The **Line** table will record the names and colour codes for each underground line.

The **Station** table will record the name and map coordinates of the station. The London Underground map, which is widely recognised to be an early masterpiece of commercial art and design, is simplified so that all lines run either horizontally, vertically, or at angles of 45°. This in turn requires station name labels to be displayed in different orientations above, below, to the left or right of the underground lines as in the examples below:



Additionally, it is sometimes necessary to print multi-word station names on two lines in order to fit the available space. Staff will be able to specify these display options when entering a station on the map. Appropriate values will be entered as **StationName** and **Position** attributes.

The **RoutePoint** table will record the connections between stations along particular underground lines by means of a series of linked lists. This allows easy calculation of the number of intermediate stations between the start and destination.

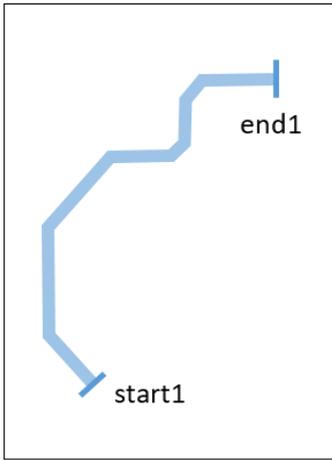


The records in the RoutePoint table will identify a station and an underground line. A **pointer** field identifies the next station along the line, whilst a **backpointer** identifies the previous station. A value of -1 will be used to mark the end of a line.

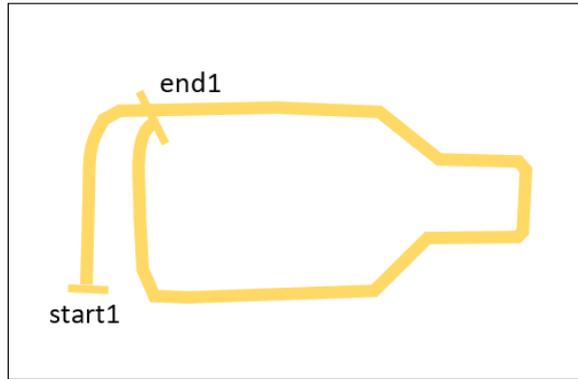
	Line	Station	Pointer	Backpointer	Branch
1	Victoria	Brixton	2	-1	1
2	Victoria	Stockwell	3	1	1
3	Victoria	Vauxhall	4	2	1
4	Victoria	Pimlico	5	3	1
5	Victoria	Victoria	6	4	1
....	.....	.....	.....	.....	.....
13	Victoria	Tottenham Hale	14	12	1
14	Victoria	Blackhorse Road	15	13	1
15	Victoria	Walthamstow Central	-1	14	1

The geometry of the network makes it necessary to identify the **branch** for each section of an underground line, so that possible routes are fully defined:

- Some underground lines run directly from start to finish, as in the case of the **Victoria line** shown below. All stations along the route will be marked as serving **branch 1**.
- The **Circle line** is unusual in forming a closed loop. Trains travel around the loop in both clockwise and anticlockwise directions. All stations will be marked as serving **branch 1**. A station appears twice in the linked list, and forms the closure point of the loop.

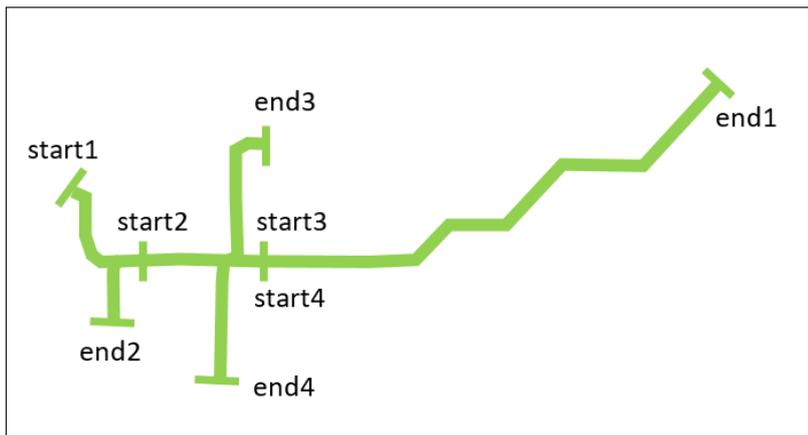


Victoria line

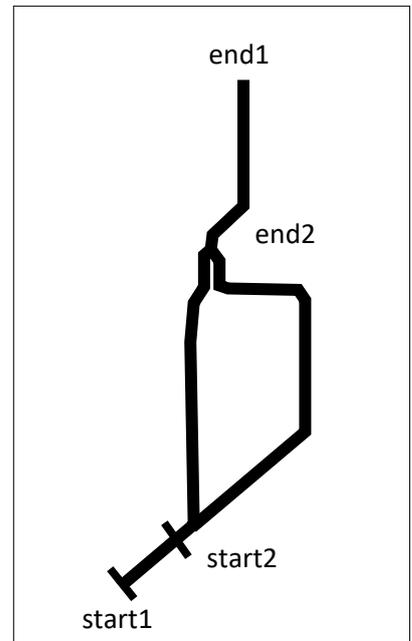


Circle line

- Some lines have multiple branches, as in the case of the **District Line**. Each of the branches will be separately numbered in the RoutePoint records, with the start and finish stations indicated by null pointer values.



District line



Northern line

- The **Northern Line** splits into two branches which then re-join after crossing central London. The branches will be separately numbered in the RoutePoint records.

Multiple underground lines may run between pairs of stations, as in the examples below. To avoid lines being hidden when the map display is drawn, each new line will be offset vertically, horizontally or diagonally when it is added. The program will specify the direction of offset by means of the **Position** attribute.

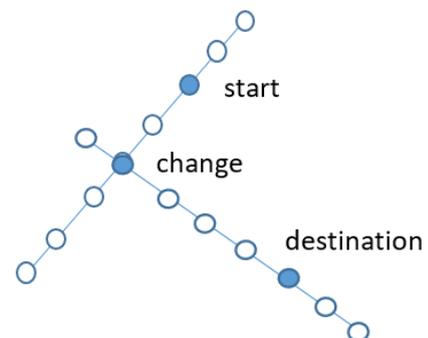


Once a series of underground lines have been recorded as linked lists, this data can be used to find routes between specified stations. The strategy which will be used is illustrated in the diagrams below.

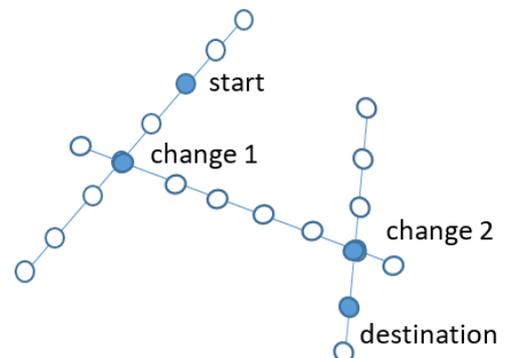
If both the start and destination stations occur on the same underground line, including any branches present, then direct travel is possible without changing train. The passenger may need to check that a train is travelling to the correct branch before boarding.



If direct travel is not possible, then a search is carried out for a route involving **one** change of train. Each station along the start line is considered as a possible change station. If another underground line passing through that station also serves the destination, then a route has been found.



If no route with a single change is found, then a search is carried out for a route involving **two** changes. Each station along the start line is considered as a possible station for **change 1**. If another underground line passes through that station, then each point along this second line is a possibility for **change 2**. If a third line passing through the change 2 station also serves the destination, then a route has been found.



If no route with two changes is found, then the search could continue in a similar way for a route with three or more changes. The interconnected nature of the London Underground system ensures that it will eventually be possible to find a route between any two specified stations.

### Programming techniques

The program will be written using **PHP** code for handling database operations and the search algorithm, whilst the **p5.js** high level extension of **JavaScript** will be used to develop interactive graphics for the map displays on the public and staff web pages.

The website will use one main page design for staff input of map data, and another public page for route finding. The pages will be largely controlled by mouse events, with some keyboard input. Data will be handled in the form of arrays of **Line**, **Station**, **RoutePoint** and **Branch** objects.

## Method

Create folders named '**underground**' on your local computer and on the internet server to store files for the project. We will begin by developing the map input system to be used by the staff. This will be password protected. Obtain an illustration which may be used on the log-in page, such as the London Underground sign shown below. Save this to the server as the file **image.jpg** with a size of approximately 450 pixels by 350 pixels.

Open a blank text document. Begin by adding the lines of code shown below. Save the file as **staffLogin.php** and copy it to the server.

```
<?
session_start();
$_SESSION['login']='NO';
?>
<html>
<head>
<title>London Underground route planning</title>
<style>
body{
font-family: arial, sans-serif;
}
</style>
</head>
<body>
<form action="staffDisplayMap.php" method="post">

<table cellpadding=20>
<tr>
<td><h3>Staff Log-in</h3>
<table border="0" cellpadding="10">
<tr>
<td>User name</td>
<td>
<?
echo "<input type=text size=20 name=user >";
?>
</td>
</tr>
<tr>
<td>Password</td>
<td>
<?
echo "<input type=password size=20 name=pass >";
?>
</td>
</tr>
<tr>
<td></td>
<td>
<input type=submit value="Enter">
</td>
</tr>
</table>
</td>
</tr>
</table>
</form>
</body>
</html>
```

Run the staffLogin page in the web browser.



**Staff Log-in**

User name

Password

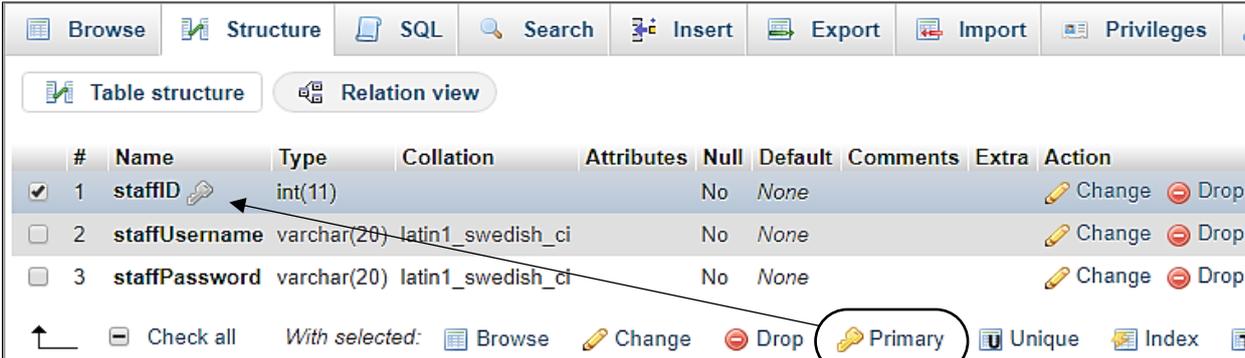
The page provides input boxes for the staff user name and password, then a button to continue. The page is arranged as a **form**, so that the input values **user** and **pass** will be transferred as variables to a page **staffDisplayMap.php** which will be loaded when the button is pressed.

A session variable '**login**' is given a value of 'NO' when the log-in sequence begins. This variable will be reset to 'YES' when a valid log-in is made. The session variable will authorise access to the staff map page and allow changes to be made to the database.

The checking of staff user names and passwords will be handled by a **Staff** object class, linked to a **staff** database table. We will begin by setting up the database table.

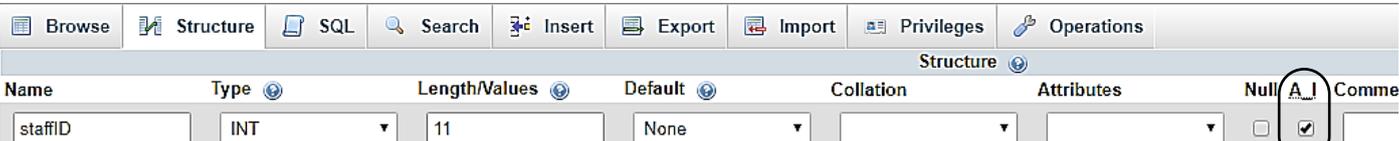
Log-in to the PHP MyAdmin web site for your database account and display the list of tables in the database. Select the **New** option from the list of tables. Name the table as '**staff**'.

Set up three fields: **staffID** as integer, **staffUsername** and **staffPassword** both of type varchar with a length of 20 characters.



#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
<input checked="" type="checkbox"/>	1	staffID		int(11)	No	None			Change  Drop
<input type="checkbox"/>	2	staffUsername	latin1_swedish_ci	varchar(20)	No	None			Change  Drop
<input type="checkbox"/>	3	staffPassword	latin1_swedish_ci	varchar(20)	No	None			Change  Drop

Click the checkbox alongside the **staffID** field, then click the **Primary** icon to set this as the primary key field of the table. Click the Change option on the **staffID** line, then tick the auto increment (**A\_I**) box:



Name	Type	Length/Values	Default	Collation	Attributes	Null	A_I	Comments
staffID	INT	11	None			<input type="checkbox"/>	<input checked="" type="checkbox"/>	

+ Options				staffID	staffUsername	staffPassword
<input type="checkbox"/>				1	Jones1	abc
<input type="checkbox"/>				2	Brown1	xyz
<input type="checkbox"/> Check all    With selected:						
					Export	

We will now set up the web page that the member of staff will access after logging-in. Open a blank file and add the program code shown below. Save the file as **staffDisplayMap.php** and copy it to the server.

```
<?
session_start();
$user=$_REQUEST['user'];
$pass=$_REQUEST['pass'];
$login=$_SESSION['login'];
if (!($_SESSION['login']=='YES'))
{
    include('Staff.php');
    if (Staff::checkPassword($user,$pass)==false)
        header('Location: staffLogin.php');
    else
        $_SESSION['login']='YES';
}
?>
<html>
<head>
    <title>London Underground route planning</title>
</head>
<body>
</body>
</html>
```

Before the page runs, the PHP `$_REQUEST` lines will obtain the values entered in the text boxes on the log-in page. This user name and password will then be verified. The remainder of the page is simply a blank page template.

The next stage is to access the staff database table to verify the data entered. Begin by setting up a **user.inc** file to authorise access to the on-line database. This has the format:

```
<?
$username="YOUR USER NAME";
$password="YOUR PASSWORD";
$database="YOUR DATABASE NAME";
?>
```

Create a blank text file and copy the lines above. Replace "YOUR USER NAME" and "YOUR PASSWORD" with the username and password which give you access to the PHP MyAdmin website. The entry for "YOUR DATABASE NAME" is normally the same as the username entered on the first line. Save this small file as **user.inc** and copy it to the server.

An object oriented approach will be used when working with the database. A **Staff** class will provide a link between the staff database table and the web pages, allowing log-in details to be verified. We will create this class file now.

Open a blank text file and add the lines of program code shown below.

```

<?
class Staff
{
    private $user;
    private $pass;
    function __construct($userSet,$passSet)
    {
        $this->user = $userSet;
        $this->pass = $passSet;
    }
    private function checkUser($userWanted,$passWanted)
    {
        if (($userWanted==$this->user)&&($passWanted==$this->pass))
            return true;
        else
            return false;
    }
    public static function checkPassword($userWanted,$passWanted)
    {
        include ('user.inc');
        $conn = new mysqli(localhost, $username, $password, $database);
        if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
        $query="SELECT * FROM staff";
        $result=mysqli_query($conn, $query);
        $num=mysqli_num_rows($result);
        mysqli_close($conn);
        $i=1;
        while ($i <= $num)
        {
            $row=mysqli_fetch_assoc($result);
            $user=$row["staffUsername"];
            $pass=$row["staffPassword"];
            $staff[$i] = new Staff($user,$pass);
            $i++;
        }
        $found=false;
        for ($i=1;$i<=$num;$i++)
        {
            $answer= $staff[$i]->checkUser($userWanted,$passWanted);
            if ($answer==true)
                $found=true;
        }
        return $found;
    }
}
?>

```

Save the file as **Staff.php** and copy it to the server.

Run the staff log-in page and test the log-in function. If a correct user name and password are entered, the blank **staffDisplayMap** page should open. If incorrect details are entered, the user should be returned directly to the log-in page.

After entering the website, a series of menu options will be available to staff members. These will allow the interactive map of the London Underground system to be constructed on screen.

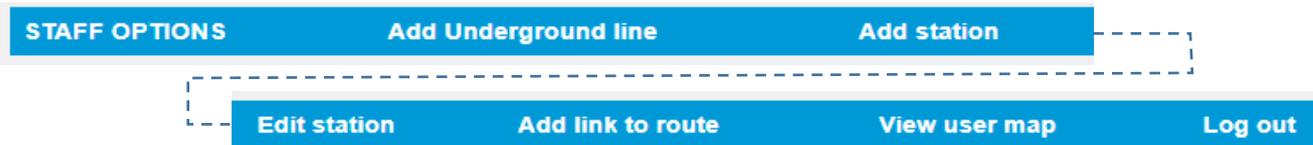
It will be convenient to create the staff menu and store the program code in a separate file. Open a blank file and add the lines of program code shown below. Save the file as **staffMenu.php** and copy it to the server.

```

<table class=menu>
  <tr>
    <th class=menu>
      <a href="staffDisplayMap.php?option=main">
        STAFF OPTIONS</a></th>
    <th class=menu>
      <a href="addLines.php?option=line">
        Add Underground line</a></th>
    <th class=menu>
      <a href="staffDisplayMap.php?option=station">
        Add station</a></th>
    <th class=menu>
      <a href="staffDisplayMap.php?option=editStation">
        Edit station</a></th>
    <th class=menu>
      <a href="addRoutes.php?option=link">
        Add link to route</a></th>
    <th class=menu>
      <a href="staffDisplayMap.php?option=map">
        View user map</a></th>
    <th class=menu>
      <a href="staffLogin.php">
        Log out</a></th>
  </tr>
</table>

```

The menu will be displayed as a bar across the top of the web page.



Before running the page, however, a style sheet will be required to provide formatting for the menu components. Open a blank file and add the lines of CSS code shown below. Save the file as **styleSheet.css** and copy it to the server.

```

body,h3,h2 {
  background-color: #F0F0F0;
  font-family: Arial, Helvetica, sans-serif;
  color: black;
  font-size: small;
}
h2 {
  font-size: 16px;
}
table.menu {
  border-collapse: collapse;
  width: 100%;
}
th.menu {
  text-align: left;
  padding: 8px;
  background-color: rgb(0, 153, 216);
  color: white;
}
a:link, a:visited {
  color: white;
  text-decoration: none;
}

```

Return to the **staffDisplayMap.php** file and add lines of code to link-in the style sheet and menu files.

```
<html>
  <head>
    <title>London Underground route planning</title>
    <link rel="stylesheet" type="text/css" href="styleSheet.css" />
  </head>
  <body>
    <?
      include('staffMenu.php');
    ?>
  </body>
</html>
```

Save the **staffDisplayMap.php** file and copy it to the server. Log-in as a member of staff and check that the menu options are displayed across the top of the page. The page area below has been given a pale grey background.



The **'Log out'** menu option closes the session and returns the user to the log-in page. Check that this works correctly, and that the user name and password must be re-entered to return to the **staffDisplayMap** page.

Obtain a digital version of the official London Underground map. The map used in this example project has a size of approximately 2500 pixels by 1900 pixels. You may choose a map which displays only the main Underground lines serving central London, or select a more comprehensive map including the Docklands Light Railway, London Overground and London Trams network. Save the map file as **tubemap.jpg** and copy it to the server.



The default screen display on entering the staff page will be this reference map. The map may also be accessed at other times by clicking the **STAFF OPTIONS** caption at the left of the menu bar.

The map will be displayed in a scrolling window which we will create using the **p5.js** high level extension of **JavaScript**. To allow p5.js to be used on the page, obtain the files

### p5.js and p5.dom.js

from the developers' web site at: [p5js.org](http://p5js.org)

Copy the files **p5.js** and **p5.dom.js** to the 'underground' folder on server.

Return to the **staffDisplayMap.php** file and add lines of code to access the p5.js and p5.dom.js files.

```
<html>
<head>
  <title>London Underground route planning</title>
  <link rel="stylesheet" type="text/css" href="styleSheet.css" />
  <script src="p5.js"></script>
  <script src="p5.dom.js"></script>
</head>
<body>
```

We will now add program code to display a section of the map. Continuing to work on the **staffDisplayMap.php** file, add the **<script>** block shown below.

```
<body>
<?
  include('staffMenu.php');
?>
<br>
<script type="text/javascript">
  var VscrollPosition=300;
  var HscrollPosition=400;
  var Hscroll=false;
  var Vscroll=false;

  function preload()
  {
    img1=loadImage("tubemap.jpg");
  }

  function setup()
  {
    createCanvas(1000, 654);
  }

  function draw()
  {
    transV = map(VscrollPosition, 0, (height-14), 0, 1890-height);
    transH = map(HscrollPosition, 0, (width-14), 0, 2560-width);
    push();
    translate(-transH, -transV);
    image(img1, 0, 0);
    pop();
  }
</script>
</body>
</html>
```

Save the **staffDisplayMap.php** file and copy it to the server. Log-in to the web site as a member of staff. A section of the map should be visible in a screen window, although scroll bars are not yet present. We will add the scroll bars next.

Several functions will be needed to operate the scrolling. It will be convenient to store these in a separate file which can be accessed by the main program.

Open a blank file and add the lines of program code shown below. The two functions create the grey scroll bars below and to the right of the map window, and add markers to indicate the scroll position.

```
<script>
function Vscrollbar(VscrollPosition)
{
    noStroke();
    fill(204);
    rect(986,0,14,640);
    fill(255);
    rect(986,640,14,14);
    fill(102);
    rect(986,VscrollPosition,14,14);
}
function Hscrollbar(HscrollPosition)
{
    noStroke();
    fill(204);
    rect(0,640,986,14);
    fill(255);
    rect(986,640,14,14);
    fill(102);
    rect(HscrollPosition,640,14,14);
}
</script>
```

Save the file as **mapFunctions.php**.

Return to the **staffDisplayMap.php** file and add a line of code to link to the mapFunctions file.

```
<body>
<?
    include('staffMenu.php');
    include('mapFunctions.php');
?>
<br>
<script type="text/javascript">
```

One further function will be required to operate the scrolling. This resets the vertical or horizontal scroll position if the mouse pointer is moved along either of the scroll bars. Add the **scrollMove( )** function shown below to the **mapFunctions.php** file, save the file and copy it to the server.

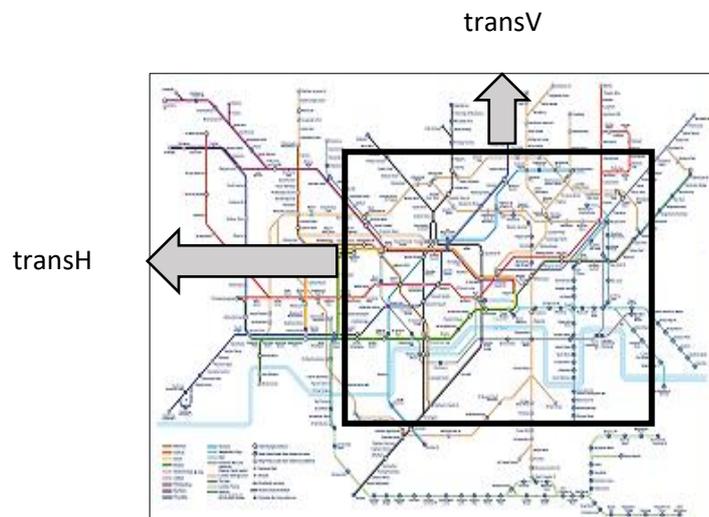
```

function scrollMove()
{
  if (((x>=960)&&(x<1020)&&(y<626))||(Vscroll==true))
  {
    if (mouseIsPressed==true)
    {
      if ((x>=960)&&(x<1020)&&(y<626))
        Vscroll=true;
      VscrollPosition=y;
      if (VscrollPosition<0)
        VscrollPosition=0;
      if (VscrollPosition>640)
        VscrollPosition=640;
    }
  }
  if (((y>=640)&&(x<976))||(Hscroll==true))
  {
    if (mouseIsPressed==true)
    {
      if((y>=640)&&(x<976))
        Hscroll=true;
      HscrollPosition=x;
      if (HscrollPosition<0)
        HscrollPosition=0;
      if (HscrollPosition>986)
        HscrollPosition=986;
    }
  }
  if (mouseIsPressed==false)
  {
    Hscroll=false;
    Vscroll=false;
  }
}

```

</script>

The final step is to return to the **staffDisplayMap.php** file and add lines of program code to operate the map scrolling. These will create the scroll bars, determine the position of the mouse, then alter the scroll position if a scroll bar is selected. This in turn changes the amount by which the map image will be offset horizontally or vertically when it is displayed in the screen window.



Add lines of code to the `draw()` function as shown below. Save the `staffDisplayMap.php` file and copy it to the server.

```
function draw()
{
    transV = map(VscrollPosition, 0, (height-14), 0, 1890-height);
    transH = map(HscrollPosition, 0, (width-14), 0, 2560-width);
    push();
    translate(-transH, -transV);
    image(img1, 0, 0);
    pop();

    Hscrollbar(HscrollPosition);
    Vscrollbar(VscrollPosition);
    x=mouseX;
    y=mouseY;
    scrollMove();
}
</script>
</body>
</html>
```

Run the website, logging-in as a member of staff. Check that the map can now be scrolled horizontally or vertically by dragging the mouse on the scroll bars.

This completes the reference map display. We can now work on the other staff options.

Options will be selected from the menu bar. The page will then be re-loaded with a value set for the variable '`option`'. The program should begin by obtaining the value of this variable.

Return to the `staffDisplayMap.php` file and add a line of program code to the PHP block at the start of the page.

```
if (Staff::checkPassword($user,$pass)==false)
    header('Location: staffLogin.php');
else
    $_SESSION['login']='YES';
}
$optionSelected = $_REQUEST['option'];
?>
<html>
<head>
```

We will then convert the PHP variable '`option`' into an equivalent JavaScript variable by means of JSON encoding. Add a line of code near the start of the `<script>` block to do this.

```
<script type="text/javascript">
    var VscrollPosition=300;
    var HscrollPosition=400;
    var Hscroll=false;
    var Vscroll=false;

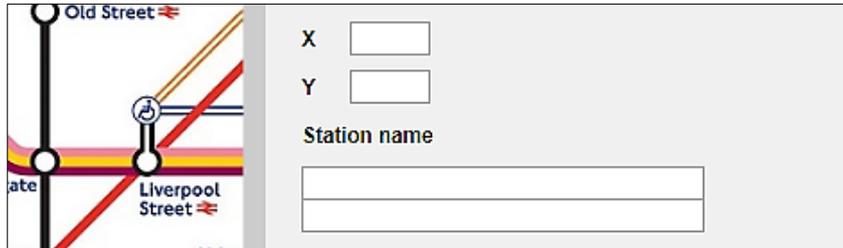
    var optionSelected= <? echo json_encode($optionSelected); ?>;

    function preload()
    {
        img1=loadImage("tubemap.jpg");
    }
```

Save the **staffDisplayMap.php** file.

When the option to **add a station** is selected, the user will click on the map to indicate the position of the station, and type the name of the station into a text input box. We will add a function to handle these operations.

Return to the **mapFunctions.php** file and add the **stationInput()** function shown below to place textbox components to the right of the map display.



```
function stationInput()
{
    inputX = createInput();
    inputX.position(1060, 170);
    inputX.size(50);
    captionX = createElement('h3', 'X');
    captionX.position(1030, 160);
    inputY = createInput();
    inputY.position(1060, 200);
    inputY.size(50);
    captionY = createElement('h3', 'Y');
    captionY.position(1030, 190);
    captionS = createElement('h3', 'Station name');
    captionS.position(1030, 220);
    inputS = createInput();
    inputS.position(1030, 260);
    inputS.size(250);
    inputS2 = createInput();
    inputS2.position(1030, 280);
    inputS2.size(250);
}
```

</script>

Save the **mapFunctions.php** file and copy it to the server.

Return to **staffDisplayMap.php** and add program code to call the **stationInput()** method.

```
function setup()
{
    createCanvas(1000, 654);
    if (optionSelected=='station')
    {
        stationInput();
    }
}
function draw()
{
```

Save the **staffDisplayMap.php** file and copy it to the server. Run the website, logging-in as a member of staff. Select 'Add station' from the menu and check that text boxes and labels are displayed to the right of the map.

Return to the **staffDisplayMap.php** file. We will now arrange for a marker to be displayed on the map when the user clicks the mouse to indicate the position of a station. Two variables **newX** and **newY** will be required. Add these near the beginning of the **<script>** block, as shown below.

```
var HscrollPosition=400;
var Hscroll=false;
var Vscroll=false;

var newX=0;
var newY=0;

var optionSelected= <? echo json_encode($optionSelected); ?>;

function preload()
{
```

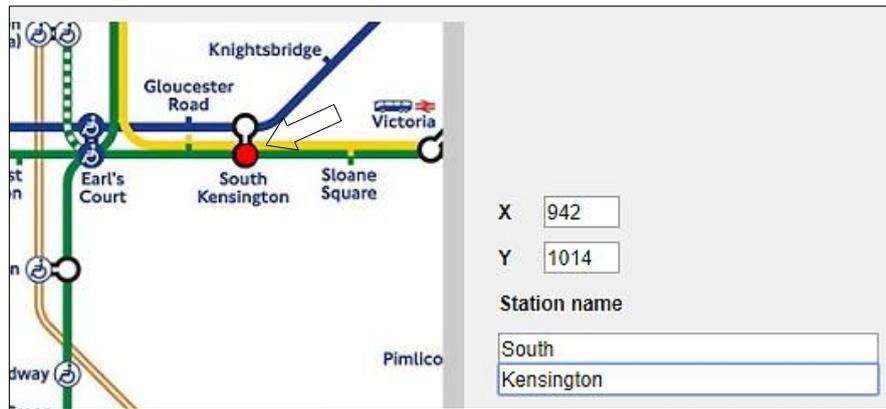
Go now to the **draw()** function and add the lines of program code shown below.

```
image(img1, 0, 0);
pop();

if (optionSelected=='station')
{
  if (mouseIsPressed==true)
  {
    if((x<940)&&(y<580))
    {
      xpos=int(x)+int(transH);
      ypos=int(y)+int(transV);
      inputX.value(xpos);
      inputY.value(ypos);
      newX=x;
      newY=y;
    }
  }
  if ((newX+newY)>0)
  {
    fill(255,0,0);
    stroke(0);
    ellipse(newX,newY,14,14);
  }
}

Hscrollbar(HscrollPosition);
Vscrollbar(VscrollPosition);
x=mouseX;
y=mouseY;
scrollMove();
}
</script>
```

Save the **staffDisplayMap.php** file and copy it to the server. Re-run the **staffDisplayMap page**, selecting the '**Add station**' option. Move the mouse pointer onto the map and click on a station. A red circle should appear at the mouse position, and the x, y coordinates will be displayed.



The station name can then be typed into the input boxes, either on a single line or on two lines as required for the map display.

Notice that it may be necessary to display the station name above, below, or to the left or right of the circular marker, depending on the pattern of the Underground lines in that particular area of the map. We will allow the user to specify the display position for the station name.

Return to the **staffDisplayMap.php** file and locate the **setup( )** function. Add lines of code as shown:

```
function setup()
{
    createCanvas(1000, 654);
    if (optionSelected=='station')
    {
        stationInput();
        buttonArray();
        caption0 = createElement('h2', '0');
        caption0.position(1200, 436);
        caption00 = createElement('h3', 'station');
        caption02 = createElement('h3', 'station2');
    }
}
```

These lines of code will use a **buttonArray( )** function to create a set of buttons which will allow the station caption to be displayed in any of eight positions around the station marker symbol.



Continuing to work on the **staffDisplayMap.php** file, go now to the **draw( )** function. Add the lines of program code below. These obtain the station name input by the user, so that it can be displayed as the caption alongside the station symbol.

```

if (optionSelected=='station')
{
    station = inputS.value();
    station2 = inputS2.value();
    caption00.html(station);
    caption02.html(station2);
    sWidth = textWidth(station)*1.15;
    sWidth2 = textWidth(station2)*1.15;

    if (mouseIsPressed==true)
    {
        if((x<940)&&(y<580))

```

Move now to the end of the `<script>` block and add the `buttonArray( )` function.

```

function buttonArray()
{
    buttonP1 = createButton('&nbsp;');
    buttonP1.position(1150, 320);
    buttonP1.mousePressed(setPosition1);
    buttonP2 = createButton('&nbsp;');
    buttonP2.position(1180, 320);
    buttonP2.mousePressed(setPosition2);
    buttonP3 = createButton('&nbsp;');
    buttonP3.position(1210, 320);
    buttonP3.mousePressed(setPosition3);
    buttonP4 = createButton('&nbsp;');
    buttonP4.position(1210, 350);
    buttonP4.mousePressed(setPosition4);
    buttonP5 = createButton('&nbsp;');
    buttonP5.position(1210, 380);
    buttonP5.mousePressed(setPosition5);
    buttonP6 = createButton('&nbsp;');
    buttonP6.position(1180, 380);
    buttonP6.mousePressed(setPosition6);
    buttonP7 = createButton('&nbsp;');
    buttonP7.position(1150, 380);
    buttonP7.mousePressed(setPosition7);
    buttonP8 = createButton('&nbsp;');
    buttonP8.position(1150, 350);
    buttonP8.mousePressed(setPosition8);
}

```

```
</script>
```

Save the `staffDisplayMap.php` file and copy it to the server.

A series of small functions will display the station name in the different possible positions, with variations depending on whether a single or double line of text is required. Return to the `mapFunctions.php` file and add the functions `setPosition1( )` .. `setPosition8( )` as shown on the two pages below.

Save the `mapFunctions.php` file and copy it to the server.

```
function setPosition1()
{
    xpos=1194-sWidth; xpos2=1194-sWidth2;
    positionNo=1;
    if (sWidth2>10) {
        caption00.position(xpos, 406);
        caption02.position(xpos2, 420);
    }
    else
        caption00.position(xpos, 420);
}

function setPosition2()
{
    xpos=1204-(sWidth/2); xpos2=1204-(sWidth2/2);
    positionNo=2;
    if (sWidth2>10) {
        caption00.position(xpos, 406);
        caption02.position(xpos2, 420);
    }
    else
        caption00.position(xpos, 420);
}

function setPosition3()
{
    positionNo=3;
    if (sWidth2>10) {
        caption00.position(1220, 406);
        caption02.position(1220, 420);
    }
    else
        caption00.position(1220, 420);
}

function setPosition4()
{
    positionNo=4;
    if (sWidth2>10) {
        caption00.position(1220,430);
        caption02.position(1220,444);
    }
    else
        caption00.position(1220,438);
}

function setPosition5()
{
    positionNo=5;
    if (sWidth2>10) {
        caption00.position(1220, 456);
        caption02.position(1220, 470);
    }
    else
        caption00.position(1220, 456);
}
```

```

function setPosition6()
{
    positionNo=6;
    xpos=1208-(sWidth/2); xpos2=1208-(sWidth/2);
    if (sWidth2>10) {
        caption00.position(xpos, 456);
        caption02.position(xpos2, 470);
    }
    else
        caption00.position(xpos, 456);
}

function setPosition7()
{
    positionNo=7;
    xpos=1194-sWidth; xpos2=1194-sWidth2;
    if (sWidth2>10) {
        caption00.position(xpos,456);
        caption02.position(xpos2,470);
    }
    else
        caption00.position(xpos,456);
}

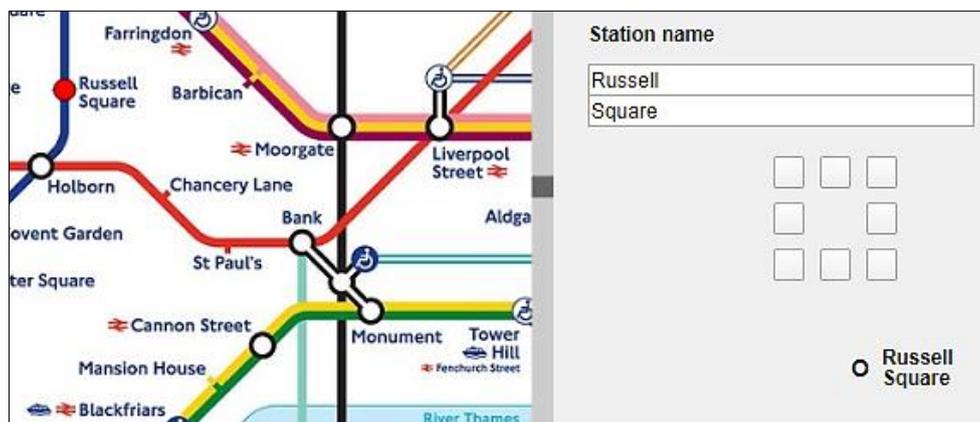
function setPosition8()
{
    positionNo=8;
    xpos=1194-sWidth; xpos2=1194-sWidth2;
    if (sWidth2>10) {
        caption00.position(xpos,430);
        caption02.position(xpos2,444);
    }
    else
        caption00.position(xpos,438);
}

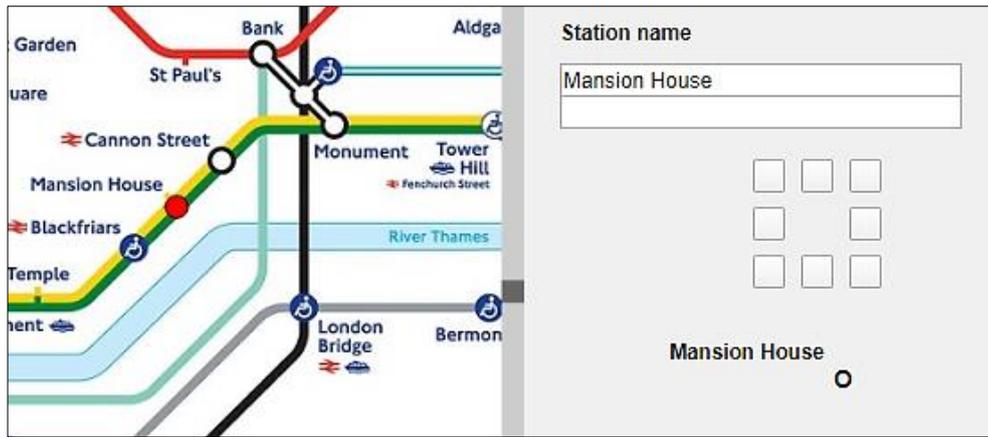
```

</script>

Save the **mapFunctions.php** file and copy it to the server. Refresh the **staffDisplayMap** page and select the **Add station** option. Carry out the sequence to enter a station, selecting the location on the map by clicking the mouse, then entering the station name on one or two input lines as required.

Check that the station caption can be moved to any of eight positions around the station symbol by clicking on the array of buttons, as in the examples below.





One further complexity needs to be considered. Major stations which serve multiple underground lines are often represented on the map by several linked symbols, as in the case of Paddington, Euston and other stations in these extracts. Only one symbol in each linked group displays the station name.



We will add a button option to hide the station name captions on the additional symbols linked within a station group. Return to the `setup()` function in the `staffDisplayMap.php` file and add the lines of program code below. Save `staffDisplayMap.php` and copy it to the server.

```

if (optionSelected=='station')
{
    stationInput();
    buttonArray();
    caption0 = createElement('h2', '0');
    caption0.position(1200, 436);
    caption00 = createElement('h3', 'station');
    caption02 = createElement('h3', 'station2');

    buttonH = createButton('hide'); buttonH.position(1030, 380);
    buttonH.mousePressed(setPosition0);
}
    
```

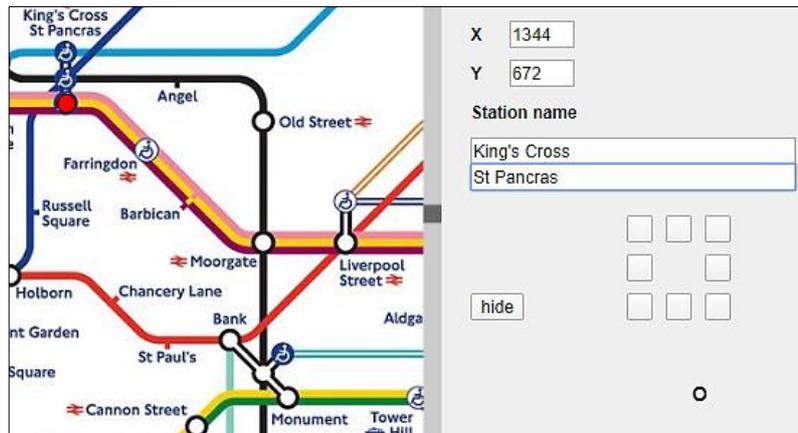
Go now to `mapFunctions.php` and add a `setPosition0()` function as shown. Save the `mapFunctions.php` file and copy it to the server.

```

function setPosition0()
{
    positionNo=0;
    caption00.position(-100, -100);
    caption02.position(-100, -100);
}
    
```

</script>

Refresh the **staffDisplayMap** web page. Enter a station name and display it in one of the eight positions around the station marker. Check that the name caption can then be hidden.



We can now arrange for the station data to be stored in a database table. Go to the PHP MyAdmin website for your database and list the existing tables. Select the 'New' option and create a table with the name '**stations**'. Add fields as shown below.

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
1	StationID	int(9)			No	None		AUTO_INCREMENT
2	StationName	varchar(40)	utf8_general_ci		Yes	NULL		
3	Xpos	int(20)			Yes	NULL		
4	Ypos	int(18)			Yes	NULL		
5	Position	int(11)			No	None		

The integer **StationID** field should be selected as the primary key, and set to auto-increment as records are added. **StationName** has the data type **varchar**, with a size of 40 characters. The remaining fields are of data type **integer**.

Data will be transferred between the web page and the database by means of **station objects**. Open a blank file and add the program code below to begin the definition of the **Stations** class.

```
<?
class Stations
{
    public static $stationObj = array();
    public $stationID;
    public $stationName;
    public $xpos;
    public $ypos;
    public $position;

    function __construct($stationID,$stationName,$xpos,$ypos,$position)
    {
        $this->stationID = $stationID;
        $this->stationName = $stationName;
        $this->xpos = $xpos;
        $this->ypos = $ypos;
        $this->position = $position;
    }
}
?>
```

The class begins with a list of attributes which correspond with the fields of the **stations** database table. The attributes are set as **public**, rather than the more usual **private** status. This is to allow the PHP Station objects to be easily converted to JavaScript objects by means of JSON encoding, for use in the interactive map display.

We will now add a method to the Stations class to save records into the database. Insert the **addStation( )** function shown below. Save the file as **Stations.php** and copy it to the server.

```

public static function addStation($stationName,$xpos,$ypos,$position)
{
    include('user.inc');
    $conn = new mysqli(localhost, $username, $password, $database);
    if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
    $query = "INSERT INTO stations VALUES ('','$stationName','$xpos',
                                                '$ypos','$position')";
    $result=mysqli_query($conn, $query);
    mysqli_close($conn);
}
}
?>

```

Return to the **staffDisplayMap.php** file and add another button to the **setup( )** function, as shown.

```

buttonH = createButton('hide'); buttonH.position(1030, 380);
buttonH.mousePressed(setPosition0);

button = createButton('add station');
button.position(1120, 520);
button.mousePressed(addStation);
}
}

```

Insert an **addStation( )** function at the end of the **<script>** block.

```

function addStation()
{
    const xloc = inputX.value();
    const yloc = inputY.value();
    result = confirm('Add this station');
    if (result==true)
    {
        window.location = "addStation.php?x="+xloc+"&y="+yloc+ "&stationName="+
            inputS.value()+"*"+inputS2.value()+"&position="+positionNo+
            "&action=add&Hscroll="+HscrollPosition+"&Vscroll="+VscrollPosition;
    }
    else
    {
        newX=0; newY=0;
    }
}
</script>

```

Save the **staffDisplayMap.php** file and copy it to the server.

A button with the caption 'add station' will be displayed below the station input components. When the button is clicked, the user will be asked to confirm that they wish to save the station record. If so, another page **addStation.php** will be loaded and the new station data transferred to it, ready for saving into the database.

Open a blank file and add the lines of program code below, then save the file as **addStation.php**. Copy the file to the server.

The **addStation** page will not be visible to the user when the web site runs. The page collects details of the station which were entered previously, then calls the method in the **Stations class** to create a new record in the database table. The user is then returned to the **staffDisplayMap** page.

```
<?
    $stationName=$_REQUEST['stationName'];
    $stationName=str_replace("'", "", $stationName);
    $xpos=$_REQUEST['x'];
    $ypos=$_REQUEST['y'];
    $position=$_REQUEST['position'];
    $Hscroll=$_REQUEST['Hscroll'];
    $Vscroll=$_REQUEST['Vscroll'];
    include ('Stations.php');
    Stations::addStation($stationName,$xpos,$ypos,$position);
    header('Location: staffDisplayMap.php?option=station&Hscroll='. $Hscroll.
        '&Vscroll='.$Vscroll);
?>
```

We can now test the input of stations. Run the **staffDisplayMap** web page and carry out the procedure for entering a station. Select the station location by clicking the mouse on the map, type the station name and use the button array to select the required display position. Finally, click the 'add station' button. A confirm dialogue box should appear. Click OK.

Enter several more stations in a similar way.

Go now to the PHP MyAdmin website and select the **stations** table. Check carefully that the stations entered on the web page have been recorded correctly in the database table. Notice that the station name includes an asterisk (\*) symbol to mark the end of the first line of text of the label, followed by a second line of text if required. For example:

- Camden Town\*      The station name should be displayed as a single line.
- Oxford\*Circus      The station name should be displayed as two separate lines.

StationID	StationName	Xpos	Ypos	Position
1	Tottenham*Court Road	1237	806	7
2	Oxford*Circus	1131	807	1
3	Camden Town*	1279	544	8
4	Swiss Cottage*	1014	588	3

The station name display **position** is indicated by an integer between 1 and 8, counting clockwise at intervals of 45° from the top-left position. A value of 0 for position indicates that the name label should be hidden.

With station data stored in the database, the next step is to reload the records and check that station symbols can be displayed to create an interactive map.

Go to the **Stations.php** class file and add a **loadStations()** method as shown below. This will create a set of PHP Station objects.

```

public static function loadStations()
{
    include ('user.inc');
    $conn = new mysqli(localhost, $username, $password, $database);
    if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
    $query="SELECT * FROM stations";
    $result=mysqli_query($conn, $query);
    $num=mysqli_num_rows($result);
    mysqli_close($conn);
    $i=1;
    while ($i <= $num)
    {
        $row=mysqli_fetch_assoc($result);
        $stationID=$row["StationID"];
        $stationName=$row["StationName"];
        $xpos=$row["Xpos"];
        $ypos=$row["Ypos"];
        $position=$row["Position"];
        $obj = new Stations($stationID,$stationName,$xpos,$ypos,$position);
        Stations::$stationObj[$i] = $obj;
        $i++;
    }
    return $num;
}
}
?>

```

Save the **Stations.php** file and copy it to the server.

Return to the **staffDisplayMap.php** file and add lines of program code to the PHP block near the start. These will access the Stations class file and create the set of Station objects.

```

if (Staff::checkPassword($user,$pass)==false)
    header('Location: staffLogin.php');
else
    $_SESSION['login']='YES';
}
$optionsSelected = $_REQUEST['option'];
include ('Stations.php');
$stationCount=Stations::loadStations();
?>
<html>
<head>
    <title>London Underground route planning</title>

```

Move down to the **<script>** block and add lines of code as shown below. These convert the PHP Station objects to an equivalent set of JavaScript Station objects.

```

var HscrollPosition=400;
var Hscroll=false;
var Vscroll=false;
var newX=0;
var newY=0;
var optionSelected= <? echo json_encode($optionSelected); ?>;

stationObj = <? echo json_encode(Stations::$stationObj); ?>;
stationCount = <? echo json_encode($stationCount); ?>;

function preload()
{
    img1=loadImage("tubemap.jpg");
}

```

Go now to the **draw( )** function and add the line of code shown below. This will call a **displayStations( )** function which will add symbols to the underground map to indicate the locations of the stations which have been recorded in the database table.

```

translate(-transH, -transV);
image(img1, 0, 0);
pop();
if (optionSelected=='station')
{
    displayStations();

    station = inputS.value();
    station2 = inputS2.value();
    caption00.html(station);
    caption02.html(station2);
}

```

Save the **staffDisplayMap.php** file and copy it to the server.

It just remains to add the function to display the stations.

Return to the **mapFunctions.php** file and add the **displayStations( )** function at the end of the <script> block, as shown below. Save **mapFunctions.php** and copy it to the server.

```

function displayStations()
{
    for (i=1;i<=stationCount ;i++ )
    {
        xCentre=stationObj[i].xpos;
        yCentre=stationObj[i].ypos;
        fill(255,120,0);
        stroke(0);
        xpos=int(xCentre)-int(transH);
        ypos=int(yCentre)-int(transV);
        ellipse(xpos,ypos,14,14);
    }
}

</script>

```

Refresh the **staffDisplayMap** web page. When the 'Add station' option is selected, all stations recorded in the database should now be marked by orange circles. Enter records for more stations and check that these locations are marked on the map.



A slight problem is that the map moves to a default scroll position when it is reloaded after saving a station record, rather than staying at the selected scroll position. This can be easily corrected.

Return to the **staffDisplayMap.php** file and add further lines of code to the PHP block near the start. When the page is reloaded, these collect the variables representing the horizontal and vertical scroll positions for the map.

```

?optionSelected = $_REQUEST['option'];
include ('Stations.php');
$stationCount=Stations::loadStations();

$Hscroll=$_REQUEST['Hscroll'];
$Vscroll=$_REQUEST['Vscroll'];

?>
<html>
<head>

```

Move down to the **<script>** block and add the lines of program code shown. The scroll position variables are first converted from PHP to JavaScript, then used to set the map scroll position.

```

var optionSelected= <? echo json_encode($optionSelected); ?>;
stationObj = <? echo json_encode(Stations::$stationObj); ?>;
stationCount = <? echo json_encode($stationCount); ?>;

Hscroll = <? echo json_encode($Hscroll); ?>;
Vscroll = <? echo json_encode($Vscroll); ?>;

function preload()
{
    img1=loadImage("tubemap.jpg");
}

function setup()
{
    if ((Hscroll>0)|| (Vscroll>0))
    {
        VscrollPosition=Vscroll;
        HscrollPosition=Hscroll;
    }

    createCanvas(1000, 654);
    if (optionSelected=='station')
    {

```

Save the **staffDisplayMap.php** file and copy it to the server. Refresh the web page and enter another station record. The map should now remain in the same scroll position when it is reloaded.

The final step in developing the 'Add station' option is to display the station name labels in the selected positions alongside the station symbols on the map. This will help us to identify any input errors.

Return to **mapFunctions.php**. Go to the end of the file and create a **displayNames( )** function. Add the lines of program code shown below.

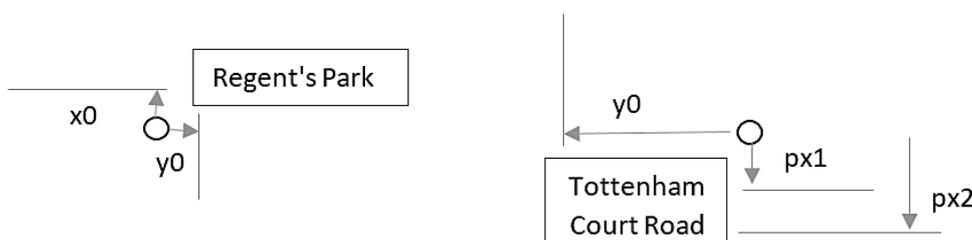
```
function displayNames()
{
  for (i=1;i<=stationCount ;i++ )
  {
    xCentre=stationObj[i].xpos;
    yCentre=stationObj[i].ypos;
    stationName=stationObj[i].stationName;
    var res = stationName.split("*");
    station=res[0];
    if(res[1]>'')
      station2=res[1];
    else
      station2='';
    widthS1=textWidth(station);
    widthS2=textWidth(station2);
    maxWidth=widthS1;
    if (widthS2>maxWidth)
      maxWidth=widthS2;
    xpos=int(xCentre)-int(transH);
    ypos=int(yCentre)-int(transV);
    pos=stationObj[i].position;
    stroke(255);
    fill(255);
  }
}
```

</script>

A loop operates for each of the stations. The x and y position of the station is found, so that a caption will be added for the station.

The station name is split into two lines of text if necessary, using the '\*' character as a divider. The pixel length of each line of text is then found so that the caption can be displayed using left, centre or right justification as appropriate.

Complete the function by inserting the lines of program code on the page below into the **displayNames( )** function. The block of code begins by setting up a series of position variables for captions in each of the eight possible positions around the station symbol. Alignment of the captions depends on whether one or two lines of text are present. The position variables are then used to draw a white background rectangle of the required size, and add the text for the station name.



```

ypos=int(yCentre)-int(transV);
pos=stationObj[i].position;
stroke(255);
fill(255);

switch (pos)
{
  case '1' : xo=-(maxWidth+10); yo=-24; y1=-24;
            px1=-(widthS1+8);
            px2=-(widthS2+8); break;
  case '2' : xo=-(maxWidth/2); yo=-26; y1=-26;
            px1=-(widthS1/2);
            px2=-(widthS2/2); break;
  case '3' : xo=8; yo=-24; y1=-24;
            px1=8; px2=8; break;
  case '4' : xo=12; yo=0; y1=-6;
            px1=12; px2=12; break;
  case '5' : xo=10; yo=20; y1=8;
            px1=10; px2=10; break;
  case '6' : xo=-(maxWidth/2); yo=22; y1=8;
            px1=-(widthS1/2);
            px2=-(widthS2/2); break;
  case '7' : xo=-(maxWidth+10); yo=20; y1=8;
            px1=-(widthS1+8);
            px2=-(widthS2+8); break;
  case '8' : xo=-(maxWidth+14); yo=-2; y1=-10;
            px1=-(widthS1+12);
            px2=-(widthS2+12); break;
}
if (int(pos)>0)
{
  if (widthS2>5)
  {
    rect(xpos+xo,ypos-12+yo,maxWidth,26);
    fill(0);
    text(station,xpos+px1,ypos+yo);
    text(station2,xpos+px2,ypos+12+yo);
  }
  else
  {
    rect(xpos+xo,ypos+y1,widthS1,16);
    fill(0);
    text(station,xpos+xo,ypos+12+y1);
  }
}
}
}
</script>

```

Save the **mapFunctions.php** file and copy it to the server.

Return to the **staffDisplayMap.php** file and add a line of program code to the **draw( )** function.

```

image(img1, 0, 0);
pop();
if (optionSelected=='station')
{
  displayStations();
  displayNames();
  station = inputS.value();
}

```



Save the **mapFunctions.php** file and copy it to the server.

Return to the **staffDisplayMap.php** file to add the 'Edit station' option. Go to the **draw( )** function and make changes to the program as shown below. Two existing lines of program code are replaced:

```
if (optionSelected=='station')
{
    displayStations();
}
```

REPLACE

The new block of code will operate for both 'Add station' and 'Edit station', but the colour of the station markers will be different in the two cases.

```
push();
translate(-transH, -transV);
image(img1, 0, 0);
pop();

if ((optionSelected=='station')||(optionSelected=='editStation'))
{
    if (optionSelected=='station')
        displayStations(255,120,0);
    if (optionSelected=='editStation')
        displayStations(0,120,255);
}

displayNames();
station = inputS.value();
station2 = inputS2.value();
```

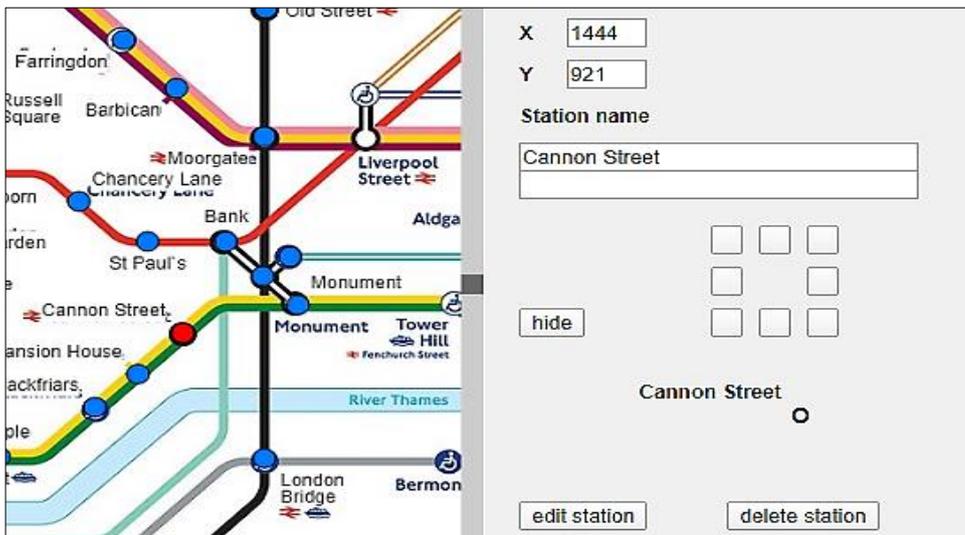
Go now to the **setup( )** function. Change the line of code shown below, so that the block of program lines will operate for both the 'Add station' and 'Edit station' options.

```
function setup()
{
    if ((Hscroll>0)||(Vscroll>0))
    {
        VscrollPosition=Vscroll;
        HscrollPosition=Hscroll;
    }
    createCanvas(1000, 654);

    if ((optionSelected=='station')||(optionSelected=='editStation'))
    {
        stationInput();
        buttonArray();
        caption0 = createElement('h2', '0');
        caption0.position(1200, 436);
    }
}
```

Save **staffDisplayMap.php** and copy it to the server and run the web page. When the 'Add station' option is selected, the entered stations should be indicated by orange symbols as before, but the 'Edit station' option should display the symbols in blue.

The objective of the 'Edit station' option is to allow the user to select a previously entered station by clicking the mouse on the map. The map coordinates, name and caption position will then be shown, and can be edited as required. Buttons will be provided to re-save the updated station record, or delete the station completely from the map.



We will begin by setting up the pair of buttons for the 'Edit station' option. Return to the `setup()` function in `staffDisplayMap.php`. Locate the block of lines which create the 'add station' button. Add an `if...else` structure as shown below, so that different buttons are created when the 'Edit station' option is selected.

```
caption02 = createElement('h3', 'station2');
buttonH = createButton('hide'); buttonH.position(1030, 380);
buttonH.mousePressed(setPosition0);

if (optionSelected=='station')
{
    button = createButton('add station');
    button.position(1120, 520);
    button.mousePressed(addStation);

}
else
{
    Ebutton = createButton('edit station');
    Ebutton.position(1030, 520);
    Ebutton.mousePressed(editStation);
    Dbutton = createButton('delete station');
    Dbutton.position(1160, 520);
    Dbutton.mousePressed(deleteStation);
}
}
```

Go now to the end of the `staffDisplayMap.php` file and add two empty functions. We will return to complete these shortly.

```
function editStation()
{
}

function deleteStation()
{
}

</script>
```

Save the **staffDisplayMap.php** file and copy it to the server. Refresh the web page and select the 'Edit station' option. Check that the edit boxes and buttons are displayed in their correct positions on the right of the map.

The next step is to transfer data into the edit boxes. Return to the **staffDisplayMap.php** file and locate the **draw( )** function. Add lines of program code as shown below, which will run a function **selectStationEdit( )** when a station is selected on the map. Save the **staffDisplayMap.php** file and copy it to the server.

```

displayNames();
station = inputS.value();
station2 = inputS2.value();
caption00.html(station);
caption02.html(station2);
swidth = textWidth(station)*1.15;
swidth2 = textWidth(station2)*1.15;

if ((optionSelected=='editStation')&&(mouseIsPressed==true)&&(x<950))
{
    selectStationEdit();
}

if (mouseIsPressed==true)
{
    if((x<940)&&(y<580))

```

Go to the **mapFunctions.php** file and begin the **selectStationEdit( )** function as shown below.

```

function selectStationEdit()
{
    for (i=1;i<=stationCount ;i++ )
    {
        xCentre=stationObj[i].xpos;
        yCentre=stationObj[i].ypos;
        Xdiff = abs(xCentre-(x+transH));
        Ydiff = abs(yCentre-(y+transV));
        if ((Xdiff<20)&&(Ydiff<20))
        {
            stationID=stationObj[i].stationID;
            stationName=stationObj[i].stationName;
            var res = stationName.split("");
            inputS.value(res[0]);
            station=res[0];
            if(res[1]>'' )
            {
                inputS2.value(res[1]);
                station2=res[1];
            }
            else
            {
                inputS2.value('');
                station2='';
            }
        }
    }
}
</script>

```

Insert a further block of code into the **selectStationEdit( )** function as shown below.

```

else
{
    inputS2.value('');
    station2='';
}

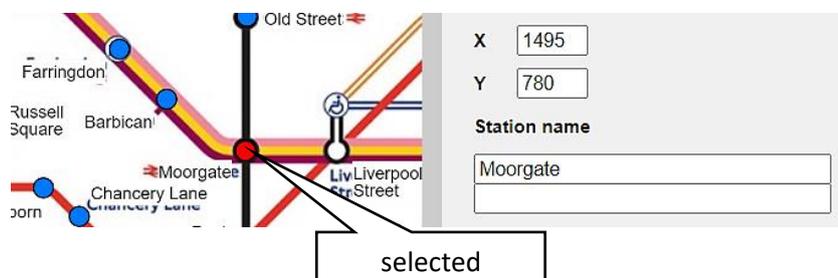
caption00.html(station);
caption02.html(station2);
swidth = textWidth(station)*1.15;
swidth2 = textWidth(station2)*1.15;
var labelPos= stationObj[i].position;
stationIDwanted = stationID;
switch(labelPos)
{
    case '0': setPosition0(); break;
    case '1': setPosition1(); break;
    case '2': setPosition2(); break;
    case '3': setPosition3(); break;
    case '4': setPosition4(); break;
    case '5': setPosition5(); break;
    case '6': setPosition6(); break;
    case '7': setPosition7(); break;
    case '8': setPosition8(); break;
}
}
}
}
</script>

```

Save the **mapFunctions.php** file and copy it to the server.

When the **selectStationEdit( )** function is called, it checks each of the station objects to determine whether any one of them lies within 20 pixels of the point where the mouse pointer was clicked. If so, the mouse pointer coordinates are transferred to the X and Y edit boxes. The station name is split into two lines of text if necessary, then written to the edit boxes. Finally, the station name is displayed as a caption in the correct position relative to the station symbol.

Run the web page, selecting the 'Edit station' option. Check that stations can be selected by clicking the mouse on the map, and the station details entered previously are displayed. It should be possible to edit the station x, y map location by dragging the red circle. The station name can be edited in the text boxes, and the label position changed by means of the array of buttons.



We can now complete the database operations for updating and deleting station records. Go to the end of the **staffDisplayMap.php** file where the empty **editStation( )** function has been inserted. Add the lines of program code shown below, which display a confirm dialogue box. If the user clicks 'OK', the program loads the **addStation.php** page, with information about the editing required included as parameters in the URL page address.

```

function editStation()
{
    const xloc = inputX.value();
    const yloc = inputY.value();
    result = confirm('Edit this station');
    if (result==true)
    {
        window.location = "addStation.php?x="+xloc+"&y="+ yloc+
            "&stationName="+inputS.value()+"*"+inputS2.value()+"&position="+
            positionNo+"&action=edit&stationID="+stationIDwanted+
            "&Hscroll="+HscrollPosition+"&Vscroll="+VscrollPosition;
    }
    else
    {
        newX=0;
        newY=0;
    }
}

```

Go now to the empty **deleteStation( )** function immediately below. Add the lines of program code shown below. A confirm dialogue box is again displayed, then the program loads the addStation.php page. Details of the station to be deleted are attached to the URL address as parameters.

Save the **staffDisplayMap.php** file and copy it to the server.

```

function deleteStation()
{
    const xloc = inputX.value();
    const yloc = inputY.value();
    result = confirm('Delete this station');
    if (result==true)
    {
        window.location = "addStation.php?x="+xloc+"&y="+ yloc+
            "&stationName="+inputS.value()+"*"+inputS2.value()+"&position="+
            positionNo+"&action=delete&stationID="+stationIDwanted+
            "&Hscroll="+HscrollPosition+"&Vscroll="+VscrollPosition;
    }
    else
    {
        newX=0;
        newY=0;
    }
}

```

Return to the **addStation.php** file. Replace the **Stations::addStation( )** command with the lines of program code below. These obtain a variable **\$action** which determines whether a station is to be added, edited or deleted. The appropriate method in the Stations class file is then run.

```

$xpos=$_REQUEST['x'];
$ypos=$_REQUEST['y'];
$position=$_REQUEST['position'];
$Hscroll=$_REQUEST['Hscroll'];
$Vscroll=$_REQUEST['Vscroll'];
include ('Stations.php');

$action=$_REQUEST['action'];
$stationID=$_REQUEST['stationID'];
if ($action=='add')
{
    Stations::addStation($stationName,$xpos,$ypos,$position);
}
if ($action=='edit')
{
    Stations::editStation($stationID,$stationName,$xpos,$ypos,$position);
}
if ($action=='delete')
{
    Stations::deleteStation($stationID);
}

header('Location: staffDisplayMap.php?option=station&Hscroll='.
                                             $Hscroll.'&Vscroll='.$Vscroll);
?>

```

Save the **addStation.php** file and copy it to the server.

Return to the **Stations.php** class file. A method has already been written to add a station. Insert the two methods below which will allow station records to be updated or deleted in the database.

```

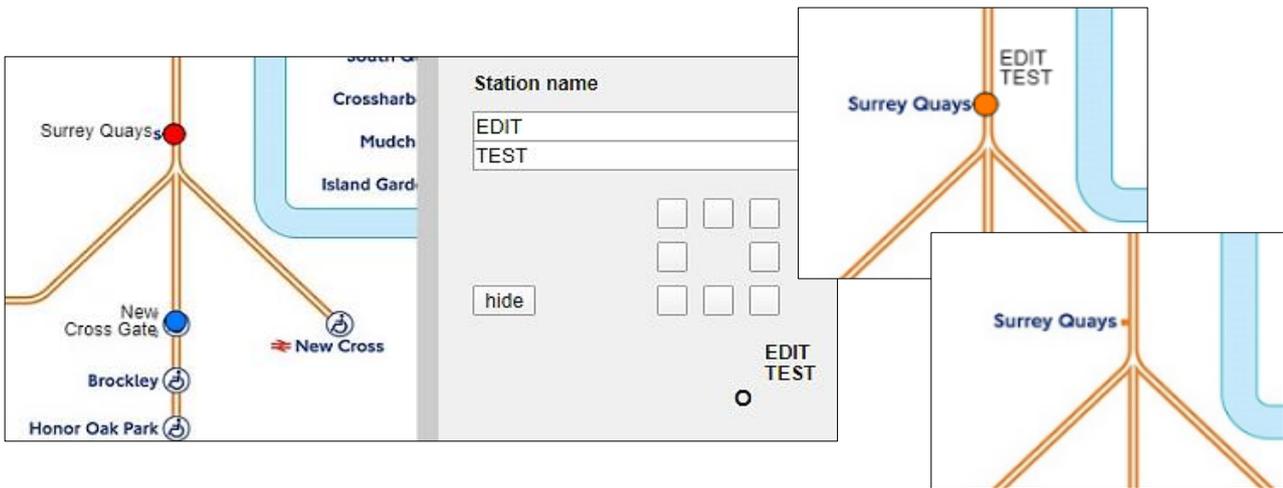
public static function editStation($stationID,$stationName,$xpos,$ypos,$position)
{
    include('user.inc');
    $conn = new mysqli(localhost, $username, $password, $database);
    if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
    $query = "UPDATE stations SET StationName='$stationName',Xpos='$xpos',
    Ypos='$ypos',Position='$position' WHERE StationID ='$stationID'";
    $result=mysqli_query($conn, $query);
    mysqli_close($conn);
}

public static function deleteStation($stationID)
{
    include('user.inc');
    $conn = new mysqli(localhost, $username, $password, $database);
    if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
    $query = "DELETE FROM stations WHERE StationID ='$stationID'";
    $result=mysqli_query($conn, $query);
    mysqli_close($conn);
}

}
?>

```

Save **Stations.php** and copy it to the server. Run the web page and select the 'Edit station' option. Check that the name text and label position can be changed for a station, and that the station can then be deleted successfully.



At this stage we can begin to set up the 'View user map' option. This will allow staff to preview the map which will be displayed to users of the website.

Return to the **staffDisplayMap.php** file and locate the **draw()** function. Replace the **image()** command and add program code as shown below. When the 'View user map' option is selected, the London Underground map image will not be displayed, but instead a white background will be set for the scroll window. The stations entered will then be plotted as grey circle symbols, along with their name captions.

```
function draw()
{
    transV = map(VscrollPosition, 0, (height-14), 0, 1890-height);
    transH = map(HscrollPosition, 0, (width-14), 0, 2560-width);
    push();
    translate(-transH, -transV);

    if (optionSelected=='map')
    {
        background(255);
    }
    else
    {
        image(img1, 0, 0);
    }

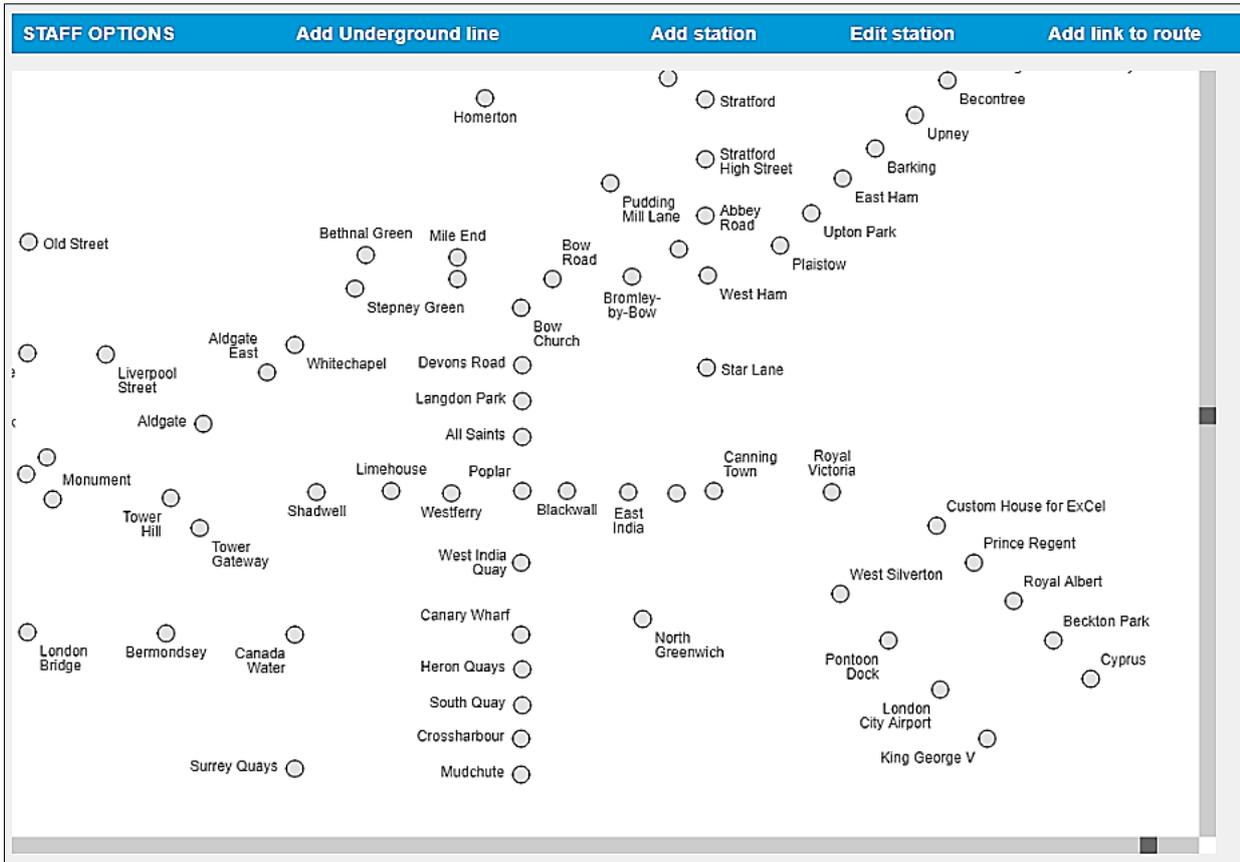
    pop();

    if (optionSelected=='map')
    {
        displayNames();
        displayStations(225,225,225);
    }

    if ((optionSelected=='station')||(optionSelected=='editStation'))
    {
        if (optionSelected=='station')
            displayStations(255,120,0);
    }
}
```

Save the **staffDisplayMap.php** file and copy it to the server. Run the web page and select the 'View user map' option. Check that the stations which you entered earlier are displayed, and that the map can be scrolled correctly, as in the image below.

We will return to complete the 'View user map' option when the routes of underground lines have been entered.



We can now move on to enter the underground lines. The first step is to set up a database table to store the names of underground lines and their colour codes for display on the map. On the official map, some lines are shown in solid colour whilst others are shown by outlines:



An additional parameter '**solid**' will be included, which can be set to **true** or **false**.

Open the PHP MyAdmin web site for your database and list the existing tables. Select the 'New' option and create a table with the name '**line**'. Insert fields as shown below. Select **lineID** to be the primary key field with data type **integer**, and set this to auto-increment as records are added. The **lineName** field is of type **varchar** with a size of 40 characters. The **colourCode** field is also set to **varchar** with a size of 12 characters. The **solid** field is of data type **boolean**.

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
1	lineID	int(11)			No	None		AUTO_INCREMENT
2	lineName	varchar(40)	latin1_swedish_ci		No	None		
3	colourCode	varchar(12)	latin1_swedish_ci		No	None		
4	solid	tinyint(1)			No	1		

The next step is to set up a **Line** class to handle the transfer of database records. Open a blank file and add the lines of program code below. These begin by defining the attributes for a Line object, which correspond with the fields of the database table. The attributes have been made **public** to allow easy conversion from PHP to an equivalent set of JavaScript objects. A constructor method is then added, along with a method to save Line records into the database.

```
<?
class Line
{
    public static $lineObj = array();
    public $lineID;
    public $lineName;
    public $colourCode;
    public $solid;

    function __construct($lineID,$lineName,$colourCode,$solid)
    {
        $this->lineID = $lineID;
        $this->lineName = $lineName;
        $this->colourCode = $colourCode;
        $this->solid = $solid;
    }

    public static function addLine($lineName,$colourCode,$solid)
    {
        include('user.inc');
        $conn = new mysqli(localhost, $username, $password, $database);
        if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
        $query = "INSERT INTO line VALUES
                    ('','$lineName','$colourCode','$solid')";
        $result=mysqli_query($conn, $query);
        mysqli_close($conn);
    }
}
?>
```

Save the file as **Line.php** and copy it to the server.

The input of underground lines and routes will take place on a new web page. Open a blank file and add the lines of program code shown in the two boxes below. Save the file as **addLines.php**.

```
<?
    $optionSelected = $_REQUEST['option'];
    include ('mapFunctions.php');
?>
<html>
<head>
    <title>London Underground route planning</title>
    <link rel="stylesheet" type="text/css" href="styleSheet.css" />
    <script src="p5.js"></script>
    <script src="p5.dom.js"></script>
</head>
<body>
<?
    include('staffMenu.php');
?>
<br>
<script type="text/javascript">
```

```

var VscrollPosition=300;
var HscrollPosition=400;
var Hscroll=false;
var Vscroll=false;

function preload()
{
    img1=loadImage("tubemap.jpg");
}

function setup()
{
    optionSelected= <? echo json_encode($optionSelected); ?>;
    createCanvas(1000, 654);
    captionL = createElement('h3', 'Line name');
    captionL.position(1050, 80);
    inputL = createInput();
    inputL.position(1050, 116);
    inputL.size(250);
    captionC = createElement('h3', 'Colour code');
    captionC.position(1050, 140);
    colourL = createColorPicker('#aa88aa');
    colourL.position(1160, 150);
    checkS = createCheckbox('solid', true);
    checkS.position(1240, 150);
    button = createButton('add line');
    button.position(1080, 200);
}
</script>
</body>
</html>

```

The page creates a scrolling window to display the underground map image in a similar way to the page where stations were entered. Input boxes are then added to allow details of underground lines to be entered.

Insert a **draw()** function underneath the **setup()** function, as shown below. This handles scrolling of the map. Save the **addLines.php** file and copy it to the server.

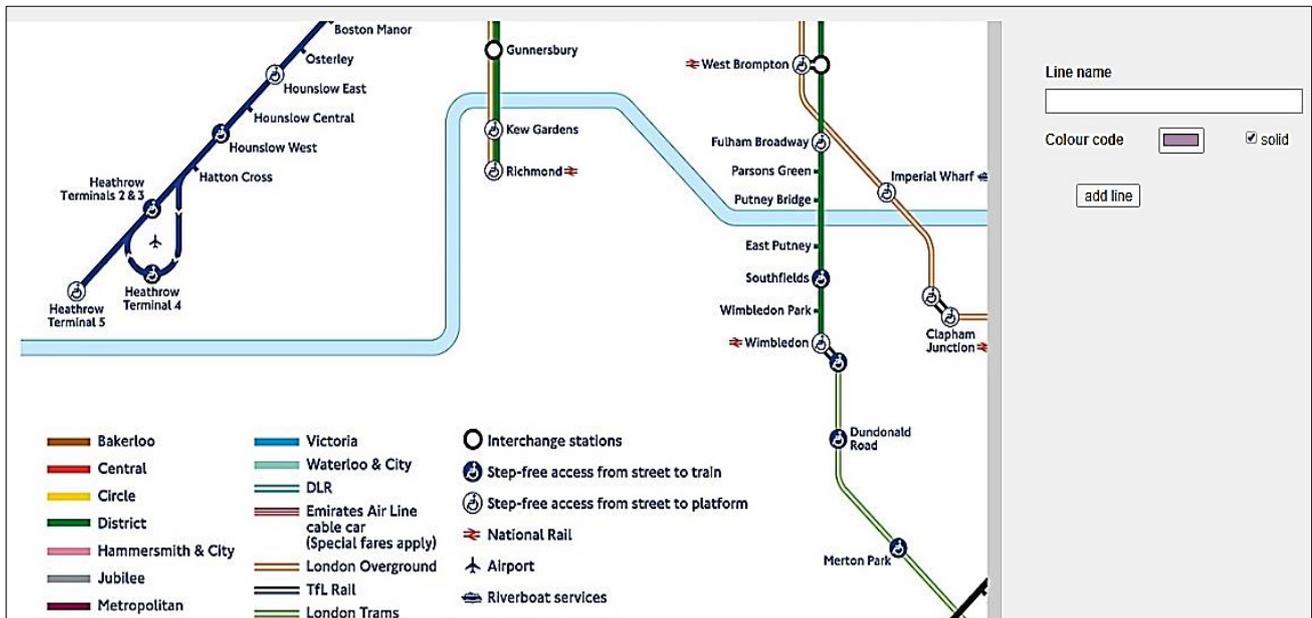
```

function draw()
{
    transV = map(VscrollPosition, 0, (height-14), 0, 1890-height);
    transH = map(HscrollPosition, 0, (width-14), 0, 2560-width);
    push();
    translate(-transH, -transV);
    image(img1, 0, 0);
    pop();
    Hscrollbar(HscrollPosition);
    Vscrollbar(VscrollPosition);
    x=mouseX;
    y=mouseY;
    scrollMove();
}

</script>
</body>
</html>

```

Run the web site, selecting the 'Add Underground line' option. The familiar scrolling map should be displayed, along with input controls including a text box for entering the Underground line name. A key to colour codes should be found in a corner of the map image.



A colour selection button is present to the right of the caption 'Colour code'. Click this button and investigate how colours may be selected from a pop-up window.



The user will enter underground line records by typing the line name, selecting the display colour for the line, and using the tick box to specify whether solid or outline colouring is to be used. We will now produce a method to save the record when the 'add line' button is clicked.

Go to the end of the the **addLines.php** file and insert the **saveLine( )** method shown below.

```

function saveLine()
{
    result = confirm('Add this line');
    if (result==true)
    {
        noStroke();
        fill(colourL.color());
        rect(0, 0, 20, 20);
        loadPixels();
        var index = (10 + (10 * width)) * 4;
        var r = pixels[index];
        var g = pixels[index+1];
        var b = pixels[index+2];
        localStorage.setItem('red', r);
        localStorage.setItem('green', g);
        localStorage.setItem('blue', b);
        window.location = "updateRoute.php?lineName="+inputL.value()+
            "&red="+r+"&green="+g+"&blue="+b+"&solid="+checkS.checked();
    }
}
</script>
</body>

```

Return to the **setup()** function of the **addLines.php** file. Insert a line of program code which will call the **saveLine()** function when the button is clicked. Save the file and copy it to the server.

```

checkS = createCheckbox('solid', true);
checkS.position(1240, 150);
button = createButton('add line');
button.position(1080, 200);

```

```

    button.mousePressed(saveLine);

```

```

}

```

Saving of the record will be controlled by another web page which we will create now. This page will not be visible to the user when the program runs. Open a blank file and add the lines of program code below.

```

<?
$lineName=$_REQUEST['lineName'];
$red=$_REQUEST['red'];
$green=$_REQUEST['green'];
$blue=$_REQUEST['blue'];
$checkS=$_REQUEST['solid'];
if ($checkS=='true')
    $solid = 1;
else
    $solid=0;
$colourCode=$red." ".$green." ".$blue;
include('Line.php');
Line::addLine($lineName,$colourCode,$solid);
header('Location: addLines.php?option=line');
?>

```

Save the file as **updateRoute.php** and copy it to the server. Run the website, selecting the 'Add Underground line' option. Enter records for several lines, specifying the line name, display colour, and whether colouring is solid or outline.

Go to the PHP MyAdmin website and select the **line** table. Check that records have been entered correctly. The colour code will be shown as three integer values in the range 0-255 which specify the red, green and blue components of the colour code. Solid line display is indicated by a value of 1, with a 0 value for outline display.

lineID	lineName	colourCode	solid
1	Central	227,44,36	1
2	Northern	36,32,33	1
3	Circle	248,210,8	1
4	Victoria	8,155,212	1

A final task is to create a key for the map which lists the Underground lines and displays their colour codes. In preparation for this, we will add another definition to the style sheet file.

Open the **styleSheet.css** file. Go to the end of the current entries and insert the style block shown below. Save **styleSheet.css** and copy it to the server.

```
div.lineTable {
    position: absolute;
    top: 300px;
    left: 1040px;
    width: 280px;
    background-color: white;
}
```

Return to the **Line.php** class file and insert a method to load the line records from the database.

```
public static function loadLines()
{
    include ('user.inc');
    $conn = new mysqli(localhost, $username, $password, $database);
    if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
    $query="SELECT * FROM line ORDER BY lineName";
    $result=mysqli_query($conn, $query);
    $num=mysqli_num_rows($result);
    mysqli_close($conn);
    $i=1;
    while ($i <= $num)
    {
        $row=mysqli_fetch_assoc($result);
        $lineID=$row["lineID"];
        $lineName=$row["lineName"];
        $colourCode=$row["colourCode"];
        $solid=$row["solid"];
        $obj = new Line($lineID,$lineName,$colourCode,$solid);
        Line::$lineObj[$i] = $obj;
        $i++;
    }
    return $num;
}
?>
```

It is convenient to also create a method in the **Line.php** class file which will display the key on the web page. Go to the bottom of the class file and add the **linelist()** method shown below. Save **Line.php** and copy it to the server.

```

public static function linelist($lineCount)
{
    echo"<div class='lineTable'>";
    echo"<table border=0>";
    for ($i=1;$i<=$lineCount; $i++)
    {
        $lineName=Line::$lineObj[$i]->lineName;
        $colourCode=Line::$lineObj[$i]->colourCode;
        $solid=Line::$lineObj[$i]->solid;
        echo " <tr height=5px >";
        echo " <td width=80px>";
        if ($solid==true)
        {
            echo"<hr size='8' style='background-color:rgb(".$colourCode.");'></td>";
        }
        else
        {
            echo"<hr style='height: 4px; border: 1px solid
                rgb(".$colourCode.");'></td>";
        }
        echo " <td width=40></td>";
        echo " <td style='font-size: 14px;'>".$lineName."</td></tr>";
    }
    echo"</table>";
    echo"</div>";
}
}
?>

```

Return to the **addLines.php** file and add program code at the beginning of the **<body>** section as shown below. This will load the Underground line records from the database then display the map key.

```

<body>
<?
    include('staffMenu.php');

    include('Line.php');
    $lineCount=Line::loadLines();
    Line::linelist($lineCount);

?>
<br>
<script type="text/javascript">

```

Save **addLines.php** and copy it to the server. Run the website, selecting the 'Add Underground line' option. It may be necessary to hold down 'CTRL' whilst clicking the refresh icon, to ensure changes to the style sheet take effect. Lines should be listed in alphabetical order, with colour codes displayed.



It is left as a programming exercise to produce *edit* and *delete* methods for Underground lines if required. We will now move on to the task of entering lines to connect stations.

Routes will be entered on a new web page. Open a blank file and add the lines of program code below. Save the file as **addRoutes.php**.

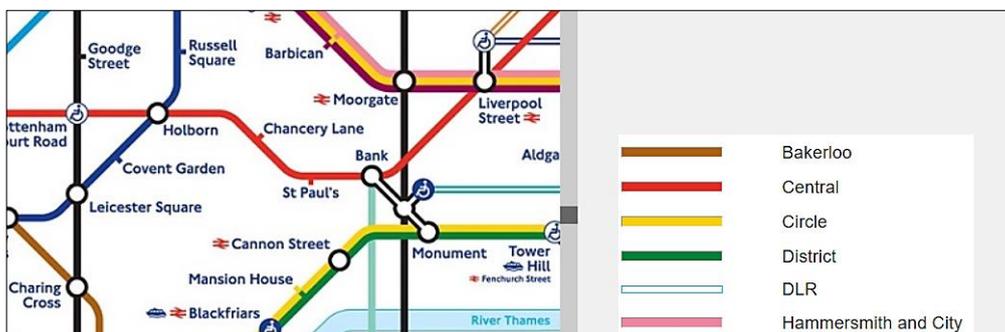
```

<html>
<head>
  <title>London Underground route planning</title>
  <link rel="stylesheet" type="text/css" href="styleSheet.css" />
  <script src="p5.js"></script>
  <script src="p5.dom.js"></script>
</head>
<body>
  <?
    include('mapFunctions.php');
    include('staffMenu.php');
    include('Line.php');
    $lineCount=Line::loadLines();
    Line::linelist($lineCount);
  ?>
</body>
</html>

```

After loading various resource files required by the page, the program displays the key panel which lists the underground lines and shows their colour codes.

Return to the **<body>** section of the file and add the lines of p5.js JavaScript code shown below. The program works in the same way as previous pages, loading the underground map image and displaying it in a scrolling window. Save the **addRoutes.php** file and copy it to the server.



```

    $lineCount=Line::loadLines();
    Line::linelist($lineCount);
?>
<br>
<script type="text/javascript">
var VscrollPosition=300;
var HscrollPosition=400;
var Vscroll=false;
var Hscroll=false;

function preload()
{
    img1=loadImage("tubemap.jpg");
}

function setup()
{
    createCanvas(1000, 654);
}

function draw()
{
    transV = map(VscrollPosition, 0, (height-14), 0, 1890-height);
    transH = map(HscrollPosition, 0, (width-14), 0, 2560-width);
    push();
    translate(-transH, -transV);
    image(img1, 0, 0);
    pop();
    Hscrollbar(HscrollPosition);
    Vscrollbar(VscrollPosition);
    x=mouseX;
    y=mouseY;
    scrollMove();
}
</script>
</body>
</html>

```

Run the website, selecting the 'Add link to route' option. Check that the scrolling map window works correctly, and that the list of underground lines is displayed to the right of the window.

Return to the **addRoutes.php** file. We will now add a drop down list which will allow the user to select an underground line along which they may enter a sequence of stations. Go to the start of the **<body>** section and add the lines of program code below. The first Line in the alphabetical list will be set as the default value.

```

<body>
<?
    include ('mapFunctions.php');
    include('staffMenu.php');
    include('Line.php');
    $lineCount=Line::loadLines();
    Line::linelist($lineCount);

    $lineWanted=Line::$lineObj[1]->lineName;
    $newLine=$_REQUEST['lineWanted'];
    if (strlen($newLine)>1)
        $lineWanted=$newLine;

?>
<br>
<script type="text/javascript">

```

Go now to the **<script>** block and add program code to the **setup()** function as shown. The lines of code begin by converting the set of PHP Line objects into an equivalent set of JavaScript objects. The names of the underground lines are then inserted into a drop down selection list. Buttons are added to allow the user to save or cancel a sequence of stations selected on the map.

```
function setup()
{
    createCanvas(1000, 654);

    lineCount= <? echo json_encode($lineCount); ?>;
    lineObj = <? echo json_encode(Line::$lineObj); ?>;
    lineWanted = <? echo json_encode($lineWanted); ?>;
    sel = createSelect();
    sel.position(1050, 80);
    sel.size(250, 30);
    sel.changed(selectEvent);
    for (i=1;i<=lineCount;i++)
    {
        sel.option(lineObj[i].lineName);
    }
    sel.selected(lineWanted);
    button = createButton('add links');
    button.position(1100, 200);
    button.mousePressed(saveLinks);
    button = createButton('clear');
    button.position(1200, 200);
    button.mousePressed(clearLinks);
}

```

When an underground line is selected from the drop-down list or one of the buttons is clicked, a JavaScript function will be called to process the request. Go to the bottom of the **<script>** block and insert the blank functions; we will return later to add program code to these.

```
Hscrollbar(HscrollPosition);
Vscrollbar(VscrollPosition);
x=mouseX;
y=mouseY;
scrollMove();
}

function selectEvent()
{
}

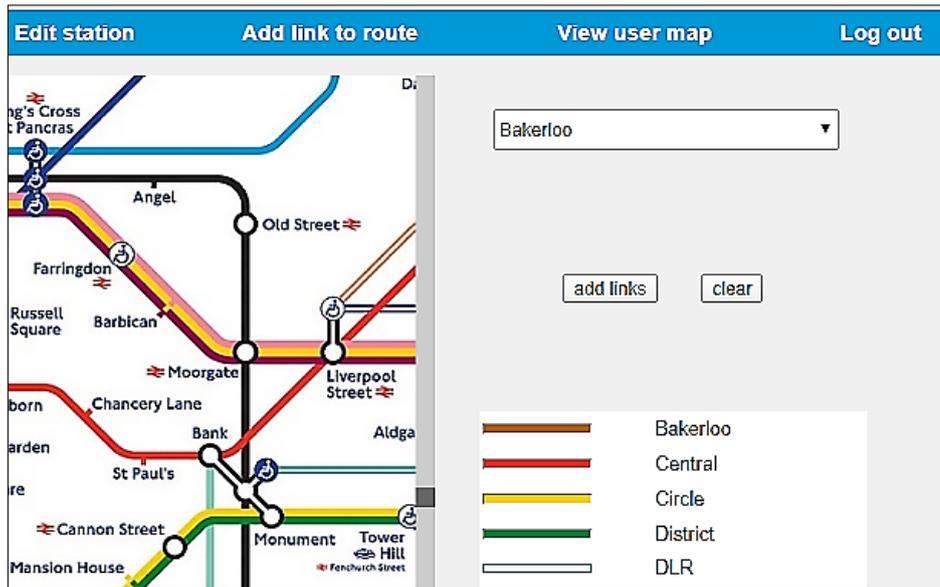
function saveLinks()
{
}

function clearLinks()
{
}

</script>
</body>
</html>

```

Save the **addRoutes.php** file and copy it to the server. Refresh the 'Add link to route' page. A drop-down selection box should list the underground lines, and two buttons have been added as in the example below.



Return to the **<body>** section of the **addRoutes.php** file and add the lines of program code shown below.

```

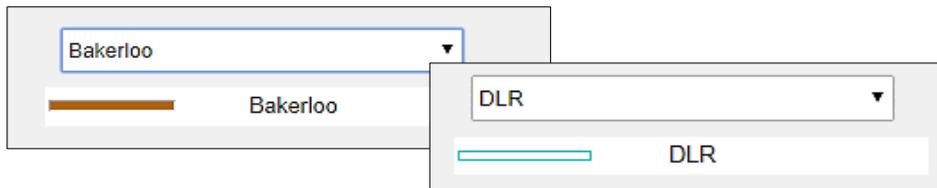
$lineWanted=Line::$lineObj[1]->lineName;
$newLine=$_REQUEST['lineWanted'];
if (strlen($newLine)>1)
    $lineWanted=$newLine;

echo"<div class='linkTable'>";
echo"<table border=0>";
for ($i=1;$i<=$lineCount; $i++)
{
    $lineName=Line::$lineObj[$i]->lineName;
    if ($lineName==$lineWanted)
    {
        $colourCode=Line::$lineObj[$i]->colourCode;
        $solid=Line::$lineObj[$i]->solid;
        echo" <tr height=5px >";
        echo" <td width=80px>";
        if ($solid==true)
        {
            echo"<hr size='8' style='background-color:rgb( ".$colourCode." );'></td>";
        }
        else
        {
            echo"<hr style='height:4px; border:1px solid rgb( ".$colourCode." );'></td>";
        }
        echo" <td width=40></td>";
        echo" <td style='font-size: 14px;'>".$lineName."</td>";
        echo"</tr>";
    }
}
echo"</table>";
echo"</div>";

?>
<br>
<script type="text/javascript">

```

We have improved the display by showing the currently selected underground line, along with its colour code, in a small panel beneath the drop-down list box.



Go now to the empty `selectEvent()` function and add a line of program code as shown. This will cause the page to be reloaded when a different underground line is selected, allowing the small panel containing the line name and colour code to be redrawn.

```
function selectEvent()
{
    window.location = "addRoutes.php?lineWanted="+sel.value();
}
```

Save the `addRoutes.php` file and copy it to the server.

Finally, open the `styleSheet.css` file and add formatting commands for the `linkTable` division as shown below. Save the `styleSheet.css` file and copy it to the server.

```
div.linkTable {
    position: absolute;
    top: 120px;
    left: 1040px;
    width: 280px;
    background-color: white;
}
```

Refresh the 'Add link to route' page. It may be necessary to hold down the CTRL key whilst clicking the page refresh icon, to ensure that the update to the style sheet has been applied to the page.

Check that the correct colour code and line name is displayed when a new selection is made from the drop-down list when the page is reloaded.



The next step is to add station markers to the map. We can make use of the `displayStations()` function stored in the `mapFunctions.php` file to do this.

Return to the `addRoutes.php` file. We will begin by loading the set of **Station objects**. Add lines of program code at the start of the `<body>` section as shown below.

```

<body>
  <?
    include ('mapFunctions.php');
    include ('Stations.php');
    $stationCount=Stations::loadStations();
    include('staffMenu.php');
    include('Line.php');
    $lineCount=Line::loadLines();
    Line::linelist($lineCount);
  >

```

Locate the **setup()** function in the **<script>** block. Add lines of program code to convert the PHP Station objects to an equivalent set of JavaScript objects.

```

function setup()
{
  createCanvas(1000, 654);
  stationObj = <? echo json_encode(Stations::$stationObj); ?>;
  stationCount = <? echo json_encode($stationCount); ?>;
  lineCount= <? echo json_encode($lineCount); ?>;
  lineObj = <? echo json_encode(Line::$lineObj); ?>;
  lineWanted = <? echo json_encode($lineWanted); ?>;
}

```

Go now to the **draw()** function and add a line of code to call the **displayStations()** function.

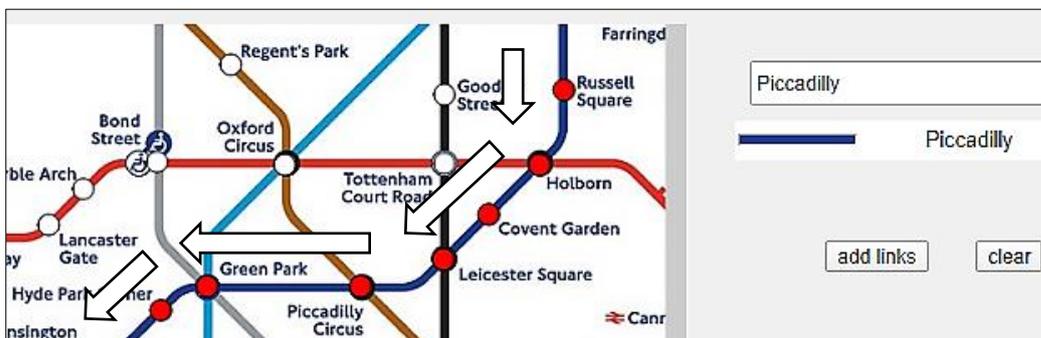
```

translate(-transH, -transV);
image(img1, 0, 0);
pop();
displayStations(255,255,255);
Hscrollbar(HscrollPosition);
Vscrollbar(VscrollPosition);

```

Save the **addRoutes.php** file and copy it to the server. Refresh the page and check that station markers are displayed as white circles.

We will now develop the user interface to allow underground routes to be entered. This will be done by dragging the mouse across a sequence of stations.



Each **Station** object will be allocated a corresponding variable in a **stationSelected[ ]** array. The array values will be initialised to **0**, indicating that no stations have been selected on the map. When the user drags the mouse over a station symbol or clicks on a symbol, the corresponding array value will be reset to **1** to indicate that the station has now been selected.

Return to the **addRoutes.php** file and locate the **<script>** block. Insert a line of code to define the `stationSelected[ ]` array.

```
var VscrollSelected=false;
var HscrollSelected=false;
var stationSelected = [];
function preload()
```

Go now to the **setup( )** function and add lines of code to initialise each element of the `stationSelected[ ]` array to 0.

```
stationObj = <? echo json_encode(Stations::$stationObj); ?>;
stationCount = <? echo json_encode($stationCount); ?>;

for (i=1;i<=stationCount ;i++ )
{
    stationSelected[i]=0;
}

lineCount= <? echo json_encode($lineCount); ?>;
```

Move now to the **draw( )** function and add the block of program code shown below.

```
image(img1, 0, 0);
pop();
displayStations(255,255,255);

if (mouseIsPressed==true)
{
    if((x<940)&&(y<580))
    {
        stationID=0;
        for (i=1;i<=stationCount ;i++ )
        {
            xCentre=int(stationObj[i].xpos);
            yCentre=int(stationObj[i].ypos);
            Xdiff = abs(xCentre-(x+transH));
            Ydiff = abs(yCentre-(y+transV));
            if ((Xdiff<10)&&(Ydiff<10))
            {
                stationID=stationObj[i].stationID;
                xCentreWanted=stationObj[i].xpos;
                yCentreWanted=stationObj[i].ypos;
                stationSelected[i]=1;
            }
        }
    }
}
for (i=1;i<=stationCount ;i++ )
{
    xCentre=int(stationObj[i].xpos);
    yCentre=int(stationObj[i].ypos);
    xpos = xCentre-transH;
    ypos = yCentre-transV;
    if (stationSelected[i]==1)
    {
        fill(255,0,0);
        ellipse(xpos,ypos,14,14);
    }
}

Hscrollbar(HscrollPosition);
```

When the user drags or clicks the mouse on the map, the program will check for any station within 10 pixels of the mouse pointer. If found, the corresponding **stationSelected[ ]** array element will be set to 1 . A program loop then checks the array and highlights all selected stations in red.

Save the **addRoutes.php** file and copy it to the server. Refresh the 'Add link to route' page. It should now be possible to select a series of stations along an underground line by clicking or dragging the mouse over the station symbols.

You may discover a couple of problems. If a series of stations are highlighted then an underground line is selected from the drop-down list, the highlighting is lost when the page reloads. To avoid this, we must retain the values in the **stationSelected[ ]** array when the page is reloaded. Another fault is that the scroll position of the map may change during the page reload. This can be avoided by retaining the values for the horizontal and vertical scroll positions.

Return to the **addRoutes.php** file and add lines of program code at the start of the **<body>** section. These reload the values for the array and scroll positions when the page is reloaded.

```

</head>
<body>
<?
    $selectString=$_REQUEST['selectString'];
    $lineChange=$_REQUEST['lineChange'];
    $Hscroll=$_REQUEST['Hscroll'];
    $Vscroll=$_REQUEST['Vscroll'];

    include ('mapFunctions.php');
    include ('Stations.php');

```

Locate the **selectEvent( )** function. Replace the **window.location** line, so that data for the selected stations and map scroll positions will be included in the URL when the page is reloaded.

```

function selectEvent()
{
    window.location = "addRoutes.php?lineWanted="+sel.value()+"&Vscroll="+
        VscrollPosition+ "&Hscroll="+HscrollPosition+"&selectString="
        +selectString+"&lineChange=YES";
}

```

A convenient way to transfer the **stationSelected[ ]** array is to convert it into a string, with each array element separated by a comma. For example, the sequence of values:

**stationSelected[1]= 0 : stationSelected[2]= 1 : stationSelected[3]= 1 : stationSelected[4]= 0**

is transferred as the string: **selectString = "0,1,1,0"**

Go to the start of the **<script>** block and add a line of program code to define the **selectString** variable.

```

var HscrollSelected=false;
var stationSelected = [];

var selectString;

function preload()

```

Move now to the **setup()** function and add the block of program code shown below. This code converts the input variables from PHP to JavaScript, the uses the scroll values to reset the position of the map. The **selectString** variable is then split at each comma to recreate the **stationSelected[]** array.

```

stationObj = <? echo json_encode(Stations::$stationObj); ?>;
stationCount = <? echo json_encode($stationCount); ?>;

selectString= <? echo json_encode($selectString); ?>;
lineChange= <? echo json_encode($lineChange); ?>;
Hscroll = <? echo json_encode($Hscroll); ?>;
Vscroll = <? echo json_encode($Vscroll); ?>;
if ((Hscroll>0)||(Vscroll>0))
{
    VscrollPosition=Vscroll;
    HscrollPosition=Hscroll;
}
if (lineChange == 'YES')
{
    stationSelected=selectString.split(",");
}
else
{
    for (i=1;i<=stationCount ;i++ )
    {
        stationSelected[i]=0;
    }

}

lineCount= <? echo json_encode($lineCount); ?>;

```

It just remains to add code to create the **selectString** variable before the page is reloaded.

Go to the **draw()** function and add the lines of program code shown.

```

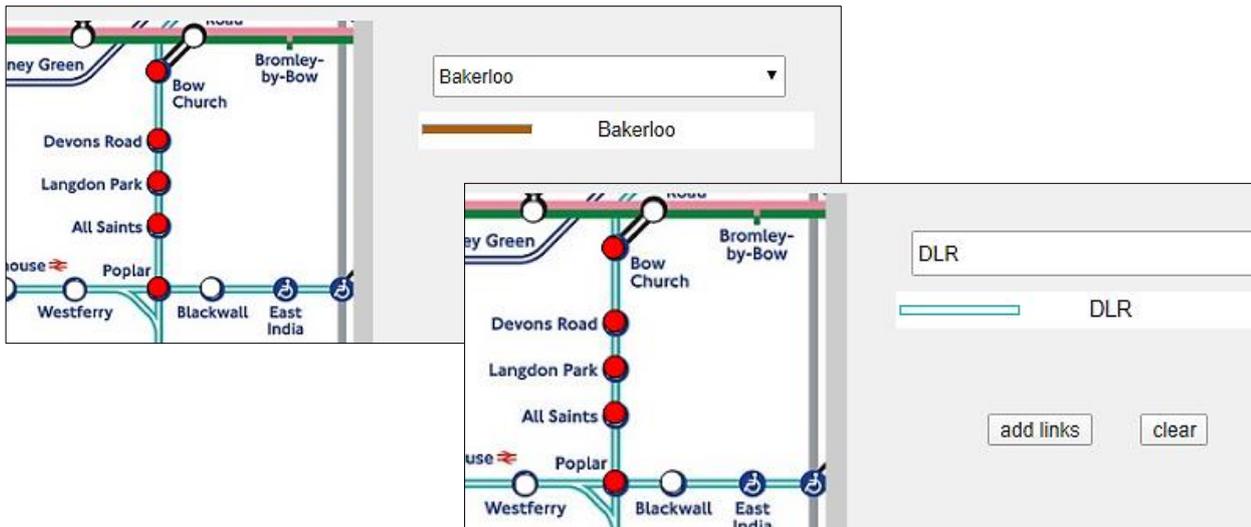
function draw()
{
    selectString='';
    for (i=1;i<=stationCount ;i++ )
    {
        selectString=selectString+', '+stationSelected[i];
    }

    transV = map(VscrollPosition, 0, (height-14), 0, 1890-height);
    transH = map(HscrollPosition, 0, (width-14), 0, 2560-width);
}

```

Save the **addRoutes.php** file and copy it to the server.

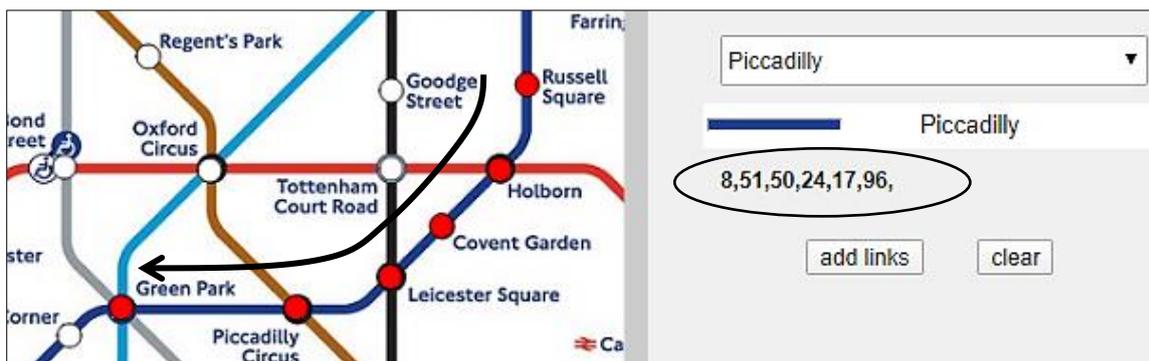
Refresh the 'Add link to route' page. Selected stations should now be retained if the line is changed, and the map should be redrawn with the scroll position unchanged, as in the example below.



A final task is to activate the 'clear' button. Return to the **addRoutes.php** file and locate the **clearLinks()** function. Add the line of program code shown below, then re-save the file and copy it to the server. Check that stations can be selected, then the selection cancelled with the 'clear' button.

```
function clearLinks()
{
    window.location = "addRoutes.php?Vscroll="+VscrollPosition
                    + "&Hscroll="+HscrollPosition;
}
```

We now have a functioning user interface and can move on to record lists of the links entered. These lists can then be stored in the database and used in calculating the most suitable journey routes. A first step will be to collect the ID numbers of stations as they are selected and build these into a string of data. The link string will be displayed on the web page for test purposes, so that we can check that stationID values are being added correctly.



Return to the **addRoutes.php** file to add the program code which will record the station sequences.

Go to the start of the **<script>** block and insert two new variables:

```
var stationSelected = [];
var selectString;
var linkString='';
var currentStation=0;

function preload()
{
```

Move up to the start of the **<body>** section and add a line of code to obtain the **linkString** value when the page is reloaded.

```
<body>
<?
$linkString=$_REQUEST['linkString'];
$selectString=$_REQUEST['selectString'];
$lineChange=$_REQUEST['lineChange'];
```

Within the **setup()** function, add lines of code as shown below. The PHP **linkString** is converted to an equivalent JavaScript variable, then displayed as a caption below the selection box on the right of the page.

```
lineCount= <? echo json_encode($lineCount); ?>;
lineObj = <? echo json_encode(Line::$lineObj); ?>;
lineWanted = <? echo json_encode($lineWanted); ?>;

linkString= <? echo json_encode($linkString); ?>;
captionC = createElement('h3', 'linkString');
captionC.position(1050, 140);
captionC.html(linkString);

sel = createSelect();
sel.position(1050, 80);
sel.size(250, 30);
```

Go next to the **draw()** function and add lines of program code as shown below.

```
if((x<940)&&(y<580))
{
    stationID=0;
    for (i=1;i<=stationCount;i++ )
    {
        xCentre=int(stationObj[i].xpos);
        yCentre=int(stationObj[i].ypos);

        if (!(i==currentStation))
        {
            Xdiff = abs(xCentre-(x+transH));
            Ydiff = abs(yCentre-(y+transV));
            if ((Xdiff<10)&&(Ydiff<10))
            {
                stationID=stationObj[i].stationID;
                xCentreWanted=stationObj[i].xpos;
                yCentreWanted=stationObj[i].ypos;
                stationSelected[i]=1;

                currentStation = i;
                if ((linkString==null)|| (linkString=='null'))
                    linkString=stationObj[currentStation].stationID+",";
                else
                    linkString += stationObj[currentStation].stationID+",";
                captionC.html(linkString);
            }
        }
    }
}
```

The p5.js programming system operates by repeating the **draw( )** function at a frame rate of 30 times per second. This allows smooth animation for scrolling the map and dragging the mouse to highlight stations. However, it is necessary to add the

```
if (!(i==currentStation))
```

condition to ensure that each new stationID is added to the link string only once, and not every time that the **draw( )** function is repeated.

Finally, go to the **selectEvent( )** function and update the window.location line, so that the linkString variable is included in the URL when the page is reloaded after selecting an underground line.

```
function selectEvent() {
    window.location = "addRoutes.php?lineWanted="+sel.value()+"&Vscroll="
                    +VscrollPosition+"&Hscroll="+HscrollPosition+
                    "&lineChange=YES&linkString="+linkString+"&selectString="+selectString;
}
```

Save the **addRoutes.php** file and copy it to the server. Run the website, logging-in as a member of staff. Click the 'Add link to route' menu option.

Select a sequence of stations, so that the station markers are highlighted in red as the mouse is dragged over them. Check that numbers are added to the link string which is displayed below the drop-down selection box for underground lines.

Go to the PHP MyAdmin web page and select the **stations** table. Check that the correct sequence of **stationID** values are now shown in the link string.

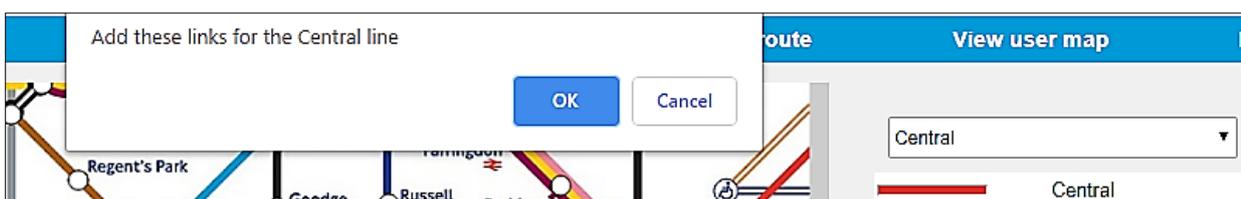
With the station selection procedure working correctly, we can now move on to store the data in the database.

Return to the **addRoutes.php** file and add lines of program code to the **saveLinks( )** function as shown. Save the **addRoutes.php** file and copy it to the server.

```
function saveLinks()
{
    result = confirm('Add these links for the '+lineWanted+' line');
    if (result==true)
    {
        window.location = "updateConnections.php?lineName="
                        +sel.value()+"&linkString="+linkString;
    }
}
```

Open a blank file and save it as **updateConnections.php**. Copy the file to the server.

Run the 'Add link to route' web page and highlight a sequence of stations along an underground line. Select the correct underground line from the drop-down list. Click the 'add links' button. A confirm dialogue box should appear.



Clicking 'OK' should take you to the blank **updateConnections.php** page. This page will handle file operations. When the program is completed it will not be visible to the user, but we can temporarily allow the page to display information for test purposes. The first piece of data needed is the ID number of the underground line. Go to **updateConnections.php** and add the following code:

```
<?
    $lineName=$_REQUEST['lineName'];
    echo"<p>lineName = ".$lineName;
    include('Line.php');
    $lineID=Line::getIDfromName($lineName);
    echo"<p>Line ID = ".$lineID;
?>
```

Save the **updateConnections.php** file and copy it to the server.

It is necessary to add another method to the **Line class** to obtain the lineID. Open the **Line.php** file and add the method shown below. Save the file and copy it to the server.

```
public static function getIDfromName($lineName)
{
    $lineCount = Line::loadLines();
    for($i=1;$i<=$lineCount;$i++)
    {
        if (Line::$lineObj[$i]->lineName == $lineName)
        {
            $lineID = Line::$lineObj[$i]->lineID;
        }
    }
    return $lineID;
}
?>
```

Return to the 'Add link to route' web page, enter a sequence of stations and select the corresponding underground line. Click the 'add links' button and select 'OK'.

The **updateConnections.php** page should open, with the name and ID number for the underground line displayed. Go to the PHP MyAdmin web page and open the **line** table. Check that the lineID value is correct.

The next step is to display the sequence of stations which were selected. Return to the **updateConnections.php** file and add the lines of program code below.

```
include('Line.php');
$lineID=Line::getIDfromName($lineName);
echo"<p>Line ID = ".$lineID;

$linkString = $_REQUEST['linkString'];
$linkArray=(explode(",",$linkString));
$linkCount= count($linkArray);
echo"<p>StationID links:";
for ($i=0;$i<$linkCount;$i++)
{
    echo"<br>".$linkArray[$i];
}
?>
```

Save the **updateConnections.php** file and copy it to the server. Refresh the web page and check that the correct sequence of stationID numbers are now listed.

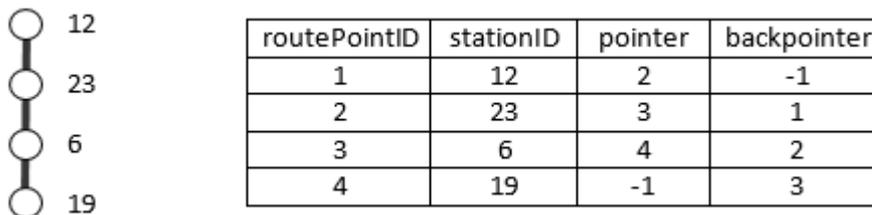
```
lineName = Piccadilly
Line ID = 16
StationID links:
8
51
50
24
17
```

We can now move on to store the sequence of stations in the database. Go to the PHP MyAdmin website for your database. List the existing tables and select the 'new' option.

Create a table with the name **routePoint**. Insert fields as shown below. All fields are of the data type **integer**. Select routePointID as the primary key, and set this to auto-increment as records are added.

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra
1	routePointID 	int(11)			No	None		AUTO_INCREMENT
2	lineID	int(11)			No	None		
3	stationID	int(11)			No	None		
4	pointer	int(11)			No	None		
5	backpointer	int(11)			No	None		
6	position	int(11)			No	None		

Route segments will be stored as linked lists. A new routePoint record will be created for each selected station and will be allocated a **routePointID** by the database, as in this example:



The station will be identified by its **stationID**. A **pointer** will be set to the routePointID of the next station along the line, with the sequence terminated by a -1 value. A **backpointer** sequence will operate in the reverse direction, indicating the previous station visited. The use of both a pointer and backpointer makes it easy to follow the station sequence in either direction.

We will create a RoutePoint class to handle data transfers between the web page and the database.

The class begins by defining the attributes for a routePoint object, which correspond to the fields of the routePoint table. A constructor method is provided, along with a method for saving routePoint records into the database.

Open a blank file and add the program code below. Save the file as **RoutePoint.php**.

```

<?
class RoutePoint
{
    public static $pointObj = array();
    public $routePointID;
    public $lineID;
    public $stationID;
    public $pointer;
    public $backpointer;
    public $position;

    function __construct($routePointID,$lineID,$stationID,
                        $pointer,$backpointer,$position)
    {
        $this->routePointID = $routePointID;
        $this->lineID = $lineID;
        $this->stationID = $stationID;
        $this->pointer = $pointer;
        $this->backpointer = $backpointer;
        $this->position = $position;
    }

    public static function addRoutePoint($lineID,$stationID,
                                        $pointer,$backpointer,$position)
    {
        include('user.inc');
        $conn = new mysqli(localhost, $username, $password, $database);
        if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
        $query = "INSERT INTO routePoint VALUES ('', '$lineID', '$stationID',
                                                '$pointer', '$backpointer', '$position')";
        $result=mysqli_query($conn, $query);
        $routePointID = mysqli_insert_id($conn);
        mysqli_close($conn);
        return $routePointID;
    }
}
?>

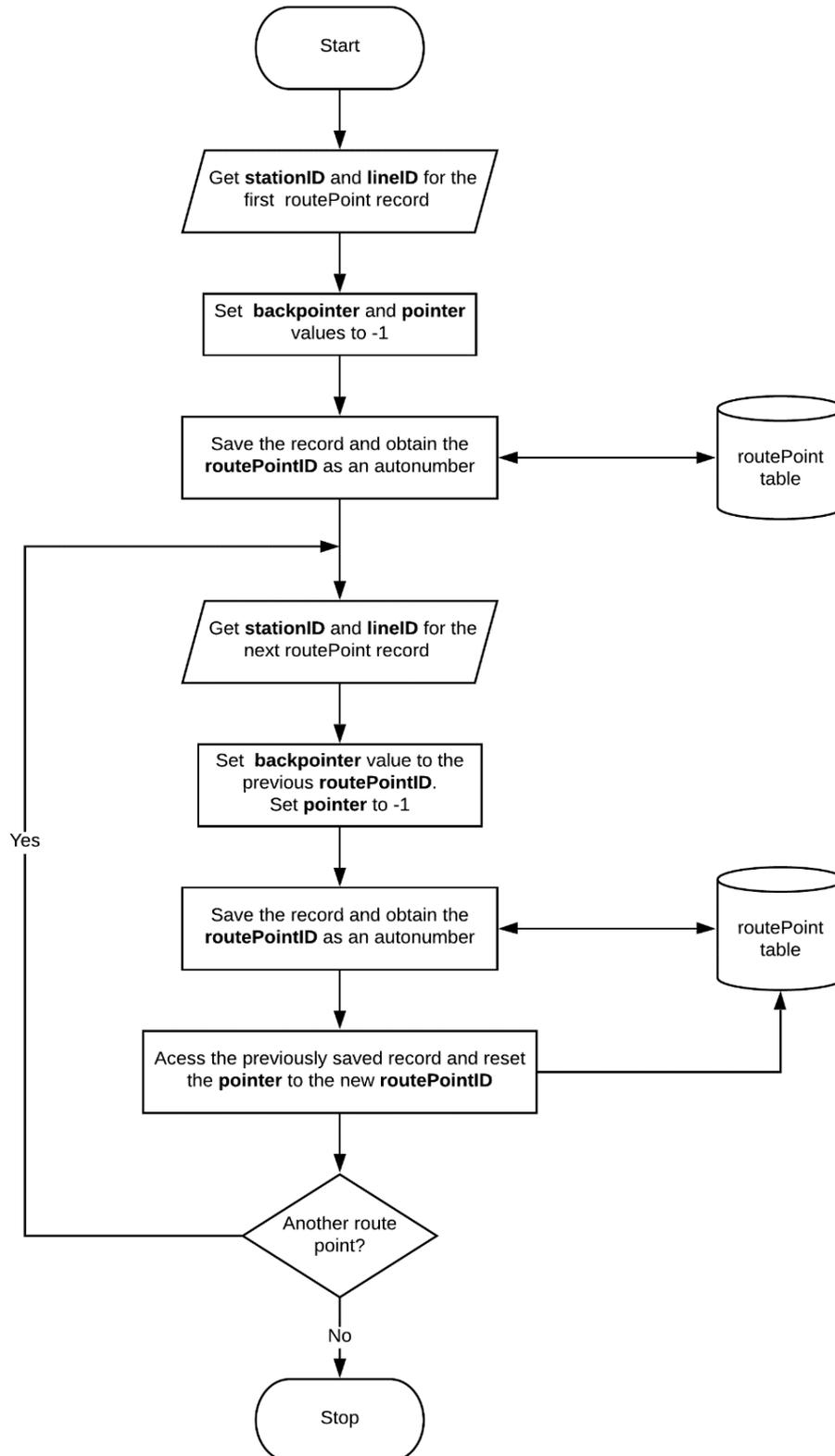
```

We can now develop an algorithm for inserting sequences of stations along an underground line. This is summarised in the flowchart below.

When the first station is entered, the **backpointer** is set to -1. The **pointer** value is not yet known, as this will depend on the routePointID which is allocated to the second station by the database auto-numbering.

When the second station has been entered, the program can return to update the **pointer** value for the first station. The **backpointer** value for the second station will be set to the routePointID of the first station, which is now known.

The process continues, with the program returning to update the **pointer** of the previous station after each new entry. When the last station is reached, the **pointer** value is set to -1.



We will add a method to the **RoutePoint.php** class file to update the **pointer** value in a routePoint record. Insert the code shown below at the end of the file. Save the file and copy it to the server.

```

public static function updatePointer($IDwanted,$pointer)
{
    include ('user.inc');
    $conn = new mysqli(localhost, $username, $password, $database);
    if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
    $query="UPDATE routePoint SET pointer='$pointer' WHERE
                                routePointID='$IDwanted'";
    $result=mysqli_query($conn, $query);
    mysqli_close($conn);
}
}

```

Return to **updateConnections.php** and add a line at the beginning to include the routePoint class.

```

<?
include('RoutePoint.php');

$lineName=$_REQUEST['lineName'];
echo"<p>lineName = ".$lineName;

```

Program code is required in the **updateConnections.php** file to implement the algorithm for uploading a station sequence to the database. Add lines to the end of the program as shown below.

```

echo"<p>StationID links:";
for ($i=0;$i<$linkCount;$i++)
{
    echo"<br>".$linkArray[$i];
}

addLinks($lineID,$linkArray);
echo"<form method=post action='addRoutes.php'>";
echo"<p><input type=submit value='continue'>";
echo"</form>";

function addLinks($lineID,$linkArray)
{
    $linkCount= count($linkArray);
    for ($i=0;$i<($linkCount-1);$i++)
    {
        $stationID=$linkArray[$i];
        $pointer=-1;
        $backpointer=-1;
        if ($i>0)
            $backpointer=$previous;
        $position=0;
        $routePointID = RoutePoint::addRoutePoint($lineID,$stationID,
                                                $pointer,$backpointer,$position);

        if ($i>0)
        {
            RoutePoint::updatePointer($previous,$routePointID);
        }
        $previous=$routePointID;
    }
}

?>

```

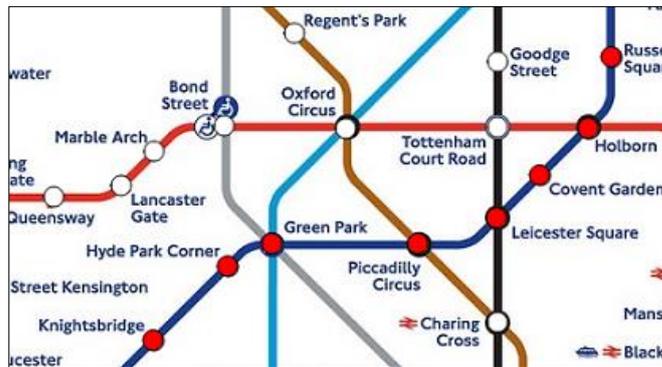
A loop operates for each of the selected stations. A record is inserted into the routePoint table by means of the method:

```
addRoutePoint(lineID, stationID, pointer, backpointer, position)
```

using a provisional value for pointer. The program then returns to update the pointer value when the route point ID for the next station has been allocated by the database. The update is carried out by:

```
updatePointer(previous routePointID, new routePointID);
```

Save the **updateConnections.php** file and copy it to the server. Run the website, log-in as a member of staff and select the 'Add link to route' option. Choose a simple underground line with no branches and no other line running alongside. Use the mouse to highlight a series of about eight stations along a mid section of the line, and select the name of the line from the drop down list.



Click the 'add links' and 'OK' buttons to reach the update connections page, then click 'continue' to return to the map display.

routePointID	lineID	stationID	pointer	backpointer	position
1	16	8	2	-1	0
2	16	51	3	1	0
3	16	50	4	2	0
4	16	24	5	3	0
5	16	17	6	4	0
6	16	96	7	5	0
7	16	280	8	6	0
8	16	54	-1	7	0

Go to the PHP MyAdmin web page and open the routePoint table. Check that a correct linked list of stations has been created. The **pointer** sequence should terminate with a -1 value, and the **backpointer** sequence should terminate with -1 when followed backwards from the last table entry. All position values are currently set to a default value of 0.

Select a sequence of stations along a central section of another simple underground line and add these to the database. Refresh the database table display. Check that the new set of routePoint records is correctly connected by pointers to produce a second linked list

The route segments just entered can now be marked on the 'Add link to route' map display. There is a problem that lines drawn with the correct colour codes will not be visible when superimposed on the map image, so a neutral colour pattern will therefore be used for all underground lines.

Return to **addRoutes.php** and add lines of code near the beginning of the file to load the route points from the database table.

```
include('mapFunctions.php');
include ('Stations.php');
$stationCount=Stations::loadStations();
```

```
include('RoutePoint.php');
$pointCount=RoutePoint::loadPoints();
```

```
include('staffMenu.php');
```

Move to the **setup()** function and add lines of program code to convert the set of PHP route point objects into an equivalent set of JavaScript objects, as shown below.

```
function setup()
{
    createCanvas(1000, 654);
    stationObj = <? echo json_encode(Stations::$stationObj); ?>;
    stationCount = <? echo json_encode($stationCount); ?>;

    pointObj = <? echo json_encode(RoutePoint::$pointObj); ?>;
    pointCount = <? echo json_encode($pointCount); ?>;

    selectString= <? echo json_encode($selectString); ?>;
    lineChange= <? echo json_encode($lineChange); ?>;
```

Move now to the **draw()** function and add a line of code to call a new function which we will add. Save the **addRoutes.php** file and copy it to the server.

```
image(img1, 0, 0);
pop();
displayStations(255,255,255);

plotLines();

if (mouseIsPressed==true)
```

Open the **mapFunctions.php** file. It will be convenient to store the graphics functions in this file so that they can be accessed by more than one web page. Add the two functions shown below, which use the ID number of a station to obtain its x,y pixel coordinates on the underground map.

```
function getXfromID(stationWanted)
{
    xpos = 0;
    for (n=1;n<=$stationCount;n++)
    {
        if (stationObj[n].stationID == stationWanted)
            xpos = stationObj[n].xpos;
    }
    return xpos;
}
function getYfromID(stationWanted)
{
    ypos = 0;
    for (n=1;n<=$stationCount;n++)
    {
        if (stationObj[n].stationID == stationWanted)
            ypos = stationObj[n].ypos;
    }
    return ypos;
}
</script>
```

Also add the **plotLines()** function shown below. Save **mapFunctions.php** and copy it to the server.

```
function plotLines()
{
    for (i=1;i<=lineCount;i++)
    {
        currentLineID=lineObj[i].lineID;
        for (j=1;j<=pointCount;j++)
        {
            if ((pointObj[j].backpointer==-1)&&(pointObj[j].lineID==currentLineID))
            {
                currentPointer=pointObj[j].pointer;
                stationIDwanted=pointObj[j].stationID;
                finished=false;
                oldX=getXfromID(stationIDwanted);
                oldY=getYfromID(stationIDwanted);
                oldX = int(oldX-transH);
                oldY = int(oldY-transV);
                count=0;
                while(finished==false)
                {
                    for (n=1;n<=pointCount;n++)
                    {
                        if (pointObj[n].routePointID==currentPointer)
                            pos=n;
                    }
                    newPointer=pointObj[pos].pointer;
                    stationIDwanted=pointObj[pos].stationID;
                    xpos=getXfromID(stationIDwanted);
                    ypos=getYfromID(stationIDwanted);
                    xpos = int(xpos-transH);
                    ypos = int(ypos-transV);
                    strokeWeight(4);
                    stroke(0);
                    line(oldX,oldY,xpos,ypos);
                    strokeWeight(2);
                    stroke(255);
                    line(oldX,oldY,xpos,ypos);
                    strokeWeight(1);
                    stroke(255,0,0);
                    line(oldX,oldY,xpos,ypos);
                    oldX=xpos;
                    oldY=ypos;
                    if (currentPointer<0)
                        finished=true;
                    currentPointer=newPointer;
                }
            }
        }
    }
}
</script>
```

Go to the **RoutePoint.php** file and add the **loadPoints()** method shown below. Save the file and copy it to the server.

```

public static function loadPoints()
{
    include ('user.inc');
    $conn = new mysqli(localhost, $username, $password, $database);
    if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
    $query="SELECT * FROM routePoint";
    $result=mysqli_query($conn, $query);
    $num=mysqli_num_rows($result);
    mysqli_close($conn);
    $i=1;
    while ($i <= $num)
    {
        $row=mysqli_fetch_assoc($result);
        $routePointID=$row["routePointID"];
        $lineID=$row["lineID"];
        $stationID=$row["stationID"];
        $pointer=$row["pointer"];
        $backpointer=$row["backpointer"];
        $position=$row["position"];
        $obj = new RoutePoint($routePointID,$lineID,$stationID,
                             $pointer,$backpointer,$position);
        RoutePoint::$pointObj[$i] = $obj;
        $i++;
    }
    return $num;
}
}

```

Run the website, logging-in as staff. Select the 'Add link to route' option. The sequences of stations entered earlier will be linked by thin red lines. The connections do not exactly follow the curves of the underground lines on the base map, but will often be sufficiently accurate for our purposes.

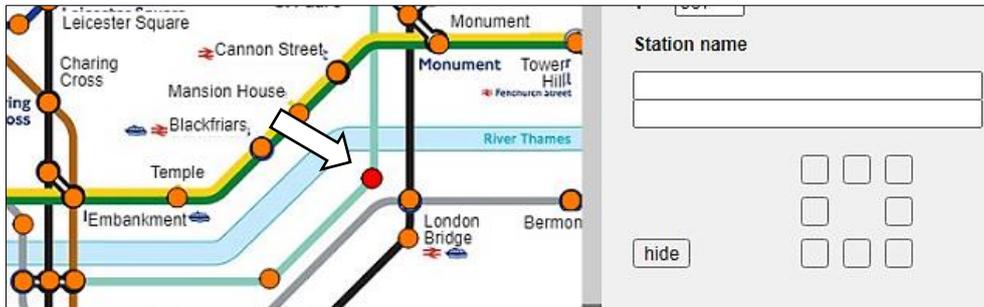


In some cases, however, stations cannot be linked satisfactorily by a direct straight line. An example is the **Waterloo and City line** which runs in a curve from Waterloo to Bank.

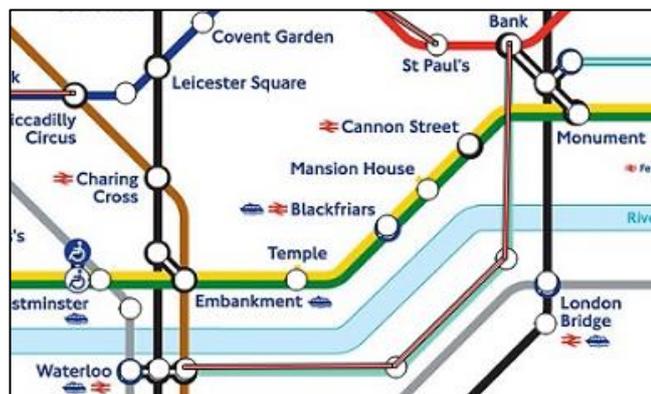


Fortunately it is quite simple to do this, as can now be demonstrated:

- Use the 'Add Underground line' menu option to set up the name and colour code for the **Waterloo and City** line.
- Go to the 'Add station' option and enter station symbols for **Waterloo** and **Bank**.
- Add station points at the two bends in the line, leaving the station name boxes blank and clicking the 'hide' button.



- Go to the 'Add link to route' menu option. Click on the sequence of points along the line from Waterloo to Bank, then enter the data. The curved line connection should appear.



Note: it is important not to add an extra curve point too close to a station, within a distance of a couple of station circle symbols, as there is a danger that the curve point may be confused with the station and cause a program error.

We will now turn our attention to the 'View user map' option. A similar procedure can be used to display underground lines, but in this case the lines will be drawn using their correct colour codes.

Open the **staffDisplayMap.php** file and add lines of code near the beginning to load the **route point** and **line** objects from the database tables.

```
include ('Stations.php');
$stationCount=Stations::loadStations();

include('RoutePoint.php');
$pointCount=RoutePoint::loadPoints();
include('Line.php');
$lineCount=Line::loadLines();

$Hscroll=$_REQUEST['Hscroll'];
```

Go to the **<script>** block and add code to convert the PHP objects to JavaScript objects, as shown below.

```

var optionSelected= <? echo json_encode($optionSelected); ?>;
stationObj = <? echo json_encode(Stations::$stationObj); ?>;
stationCount = <? echo json_encode($stationCount); ?>;

pointObj = <? echo json_encode(RoutePoint::$pointObj); ?>;
pointCount = <? echo json_encode($pointCount); ?>;
lineObj = <? echo json_encode(Line::$lineObj); ?>;
lineCount = <? echo json_encode($lineCount); ?>;

Hscroll = <? echo json_encode($Hscroll); ?>;

```

Add a line of code to the **draw()** function as shown below.

```

pop();
if (optionSelected=='map')
{
    plotLines2();
    displayNames();
    displayStations(225,225,225);
}

```

Save the **staffDisplayMap.php** file and copy it to the server.

Return to the **mapFunctions.php** file and add the **plotLines2()** function shown below and continued on the following page. This is similar to the **plotLines()** function created earlier, but this version accesses the colour codes for each underground line and sets the line display colour accordingly.

Save the **mapFunctions.php** file and copy it to the server.

```

function plotLines2()
{
    for (i=1;i<=lineCount;i++)
    {
        stroke(255);
        fill(0);
        currentLineID=lineObj[i].lineID;
        colourCode=lineObj[i].colourCode;
        solid=lineObj[i].solid;
        let c = split(colourCode, ',');
        r=int(c[0]); g=int(c[1]); b=int(c[2]);
        fill(r,g,b);
        stroke(r,g,b);
        for (j=1;j<=pointCount;j++)
        {
            if ((pointObj[j].backpointer==-1)&&(pointObj[j].lineID==currentLineID))
            {
                currentPointer=pointObj[j].pointer;
                stationIDwanted=pointObj[j].stationID;
                finished=false;
                oldX=getXfromID(stationIDwanted);
                oldY=getYfromID(stationIDwanted);
                oldX = int(oldX-transH);
                oldY = int(oldY-transV);
                finished=false;
                count=0;
            }
        }
    }
}

```

continued below...



Go to the **mapFunctions.php** file and locate the **displayStations( )** function. Modify the title line of the function to include an additional parameter **'midpoints'** as shown below. This can be set to 'YES' or 'NO', depending on whether the mid points are to be marked by circle symbols or left blank. Add the **'station name = '** line and modify the **'if...'** line as shown. Save the **mapFunctions.php** file and copy it to the server.

```
function displayStations(r,g,b,midpoints)
{
  for (i=1;i<=stationCount ;i++ )
  {
    xCentre=stationObj[i].xpos;
    yCentre=stationObj[i].ypos;
    show=true;

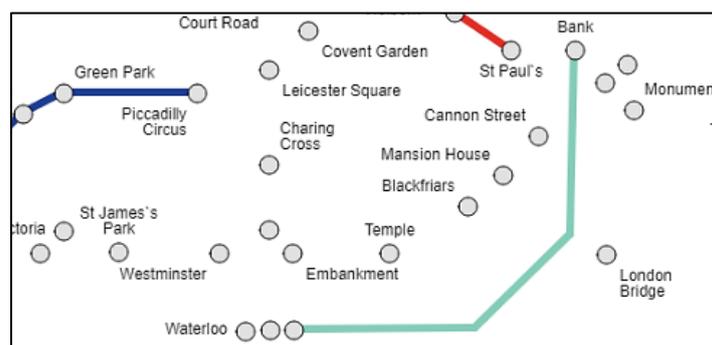
    stationName=stationObj[i].stationName;
    if ((textWidth(stationName)<8)&&(midpoints=='NO'))
    {
      show=false;
    }
    if (show==true)
    {
      fill(r,g,b);
      stroke(0);
      xpos=int(xCentre)-int(transH);
      ypos=int(yCentre)-int(transV);
      ellipse(xpos,ypos,14,14);
    }
  }
}
```

Return to the **staffDisplayMap.php** file locate the **draw( )** function. Add the extra parameter to the **displayStations( )** function call, setting the value to 'NO' as shown below.

```
if (optionSelected=='map')
{
  plotLines2();
  displayNames();

  displayStations(225,225,225,'NO');
}
```

Save **staffDisplayMap.php** and copy it to the server. Re-run the 'View user map' option. Intermediate points on the Waterloo and City line should not be marked by circles:



The key showing the colour codes for the different lines can now be added. Return to **staffDisplayMap.php** and add lines of program near the beginning of the file to do this.

```
include('Line.php');
$lineCount=Line::loadLines();
$Hscroll=$_REQUEST['Hscroll'];
$Vscroll=$_REQUEST['Vscroll'];

if ($optionSelected=='map')
{
    Line::linelist($lineCount);
}

?>
<html>
<head>
```

Save the **staffDisplayMap.php** file and copy it to the server. Run the website, selecting the 'View user map' option. Check that the list of underground lines and their colour codes is shown to the right of the map.

It is important that stations are linked in a correct sequence along each line before calculating the shortest route between the starting point and destination. A test procedure will now be added.

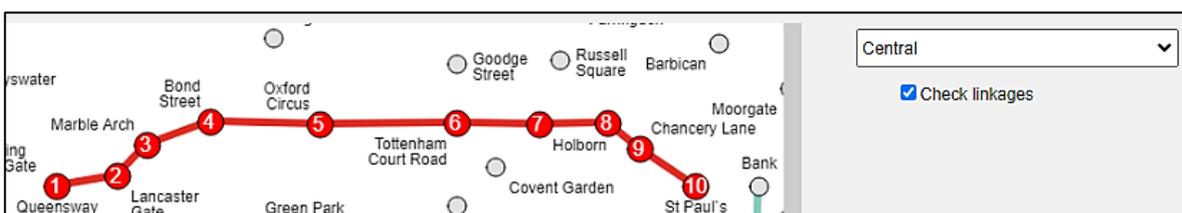
Go to the **staffDisplayMap.php** file and locate the **setup()** function. Add the lines of program code shown below.

```
function setup()
{
    if ((Hscroll>0)|| (Vscroll>0))
    {
        VscrollPosition=Vscroll;
        HscrollPosition=Hscroll;
    }
    createCanvas(1000, 654);

    lineWanted='Bakerloo';
    if (optionSelected=='map')
    {
        sel = createSelect();
        sel.position(1050, 60);
        sel.size(250, 30);
        for (i=1;i<=lineCount;i++)
        {
            sel.option(lineObj[i].lineName);
        }
        checkbox = createCheckbox('Check linkages', false);
        checkbox.position(1080, 100);
    }

    if ((optionSelected=='station')|| (optionSelected=='editStation'))
```

The test procedure will number the route points in the order in which they are connected by the linked list, as in this example for part of the Central Line:



Begin by creating a drop down list for selecting an underground line, and a tick box to activate the checking procedure. Move to the **draw( )** function and add the block of code shown below.

```

if (optionSelected=='map')
{
    plotLines2();
    displayNames();
    displayStations(225,225,225,'NO');

    lineWanted = sel.value();
    if (checkbox.checked())
    {
        routeTest(lineWanted,lineObj,lineCount);
    }
}

```

The program will obtain the name of the selected underground line from the drop-down list box. If the checkbox is ticked, a procedure **routeTest( )** will display the numbered sequence of points along the route.

Go to the end of the **staffDisplayMap.php** file and enter the **routeTest( )** function shown in the two boxes below. Save the file and copy it to the server.

```

function routeTest(lineWanted,lineObj,lineCount)
{
    var p;
    var finished=false;
    lineIDwanted = 0;
    for (i=1;i<=lineCount;i++)
    {
        if (lineObj[i].lineName == lineWanted)
            lineIDwanted = lineObj[i].lineID;
    }
    for(var i=1;i<=pointCount;i++)
    {
        if ((pointObj[i].backpointer == -1)&&
            (pointObj[i].lineID == lineIDwanted))
        {
            count=1;
            current=pointObj[i].stationID;
            next=pointObj[i].pointer;
            finished=false;
            p=i;
            while (finished==false)
            {
                x=getStationX(current);
                y=getStationY(current);
                xpos=x-int(transH);
                ypos=y-int(transV);
                fill(255,0,0);
                stroke(0);
                ellipse(xpos,ypos,20,20);
                fill(255);
                stroke(255,0,0);
                textSize(16);
            }
        }
    }
}

```

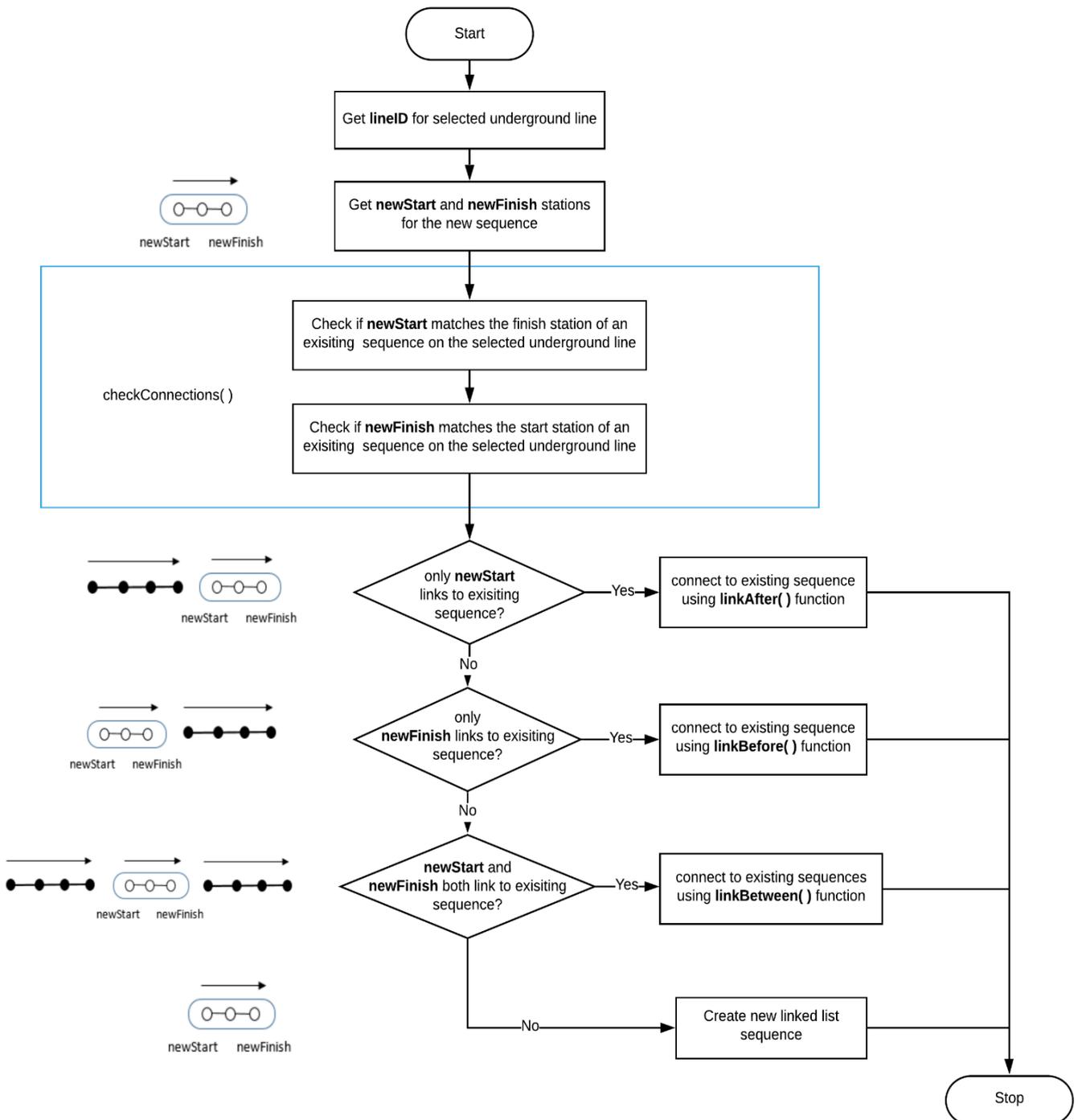


The **getStationX( )** and **getStationY( )** functions take a **stationID** as the input parameter and return the map coordinates for the station. The **getNextArrayPosition( )** function takes a **routePointID** as the input parameter, and finds the location of an object with this value in the array of routePoint objects.

Save the **mapFunctions.php** file and copy it to the server. Run the website, and go to the 'View user map' option.

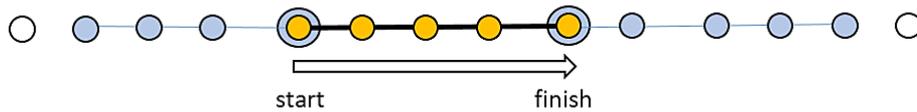
Select an underground line from the drop-down list for which a sequence of stations has been entered, then tick the checkbox. The stations should be numbered in the order in which they were entered.

We will now return to the entry of sections of underground line. Due to the length of each underground line, the user may wish to enter the route in a series of sections. An overall strategy is summarised in the flowchart below. We will work through each of the options in the following sections.



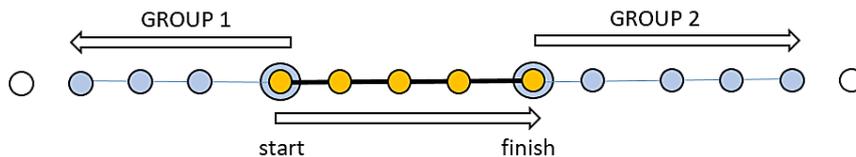
It is useful to summarise the procedures which are used to add underground lines to the map.

A new sequence of stations is selected. The first step is then to compare the start and finish stations of the new sequence with the end points of any previously entered sections on the same underground line.



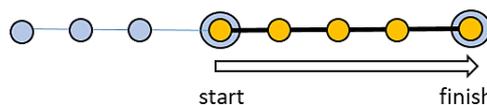
- If no corresponding end point is found, then the new section is entered as a simple linked list.
- If a previously entered route section matches the **start** of the new section, then a single combined linked list will be created by attaching the new sequence to the end of the existing station group.
- If a previously entered route section matches the **finish** of the new section, then a single combined linked list will be created by attaching the new sequence to the beginning of the existing station group.
- If previously entered route sections match both the **start** and **finish** of the new section, then a single combined linked list will be created by attaching the new sequence to the end of the first station group and the beginning of the second station group.

For the final sequence to be linked correctly, it is necessary for each group of stations to be linked sequentially in the same direction along the underground line. In this example:

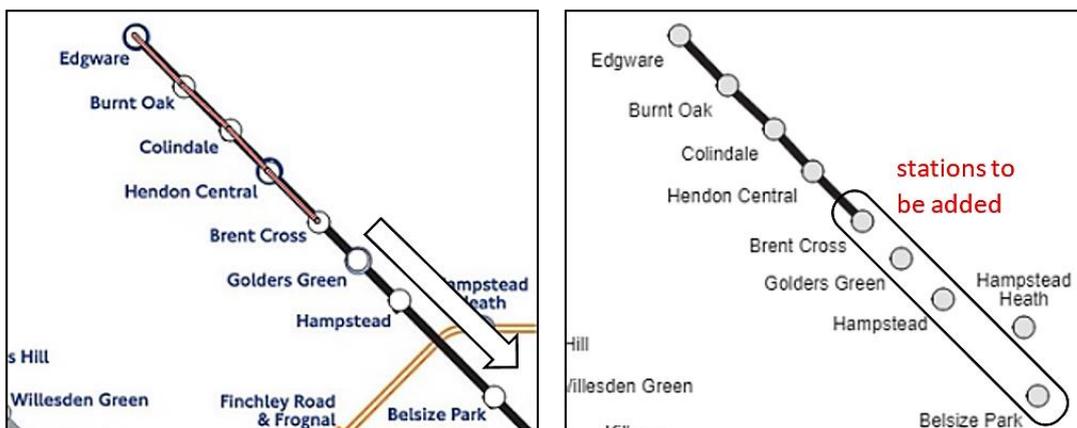


the station sequence in group 1 would have to be reversed before the linked lists are joined. The sequence in group 2 is already running in the correct direction and can be linked directly.

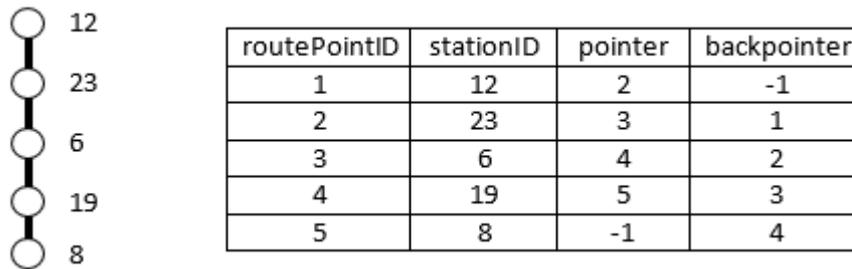
We will first consider the case of a group of stations which are to be attached to the end of a sequence which had been entered previously:



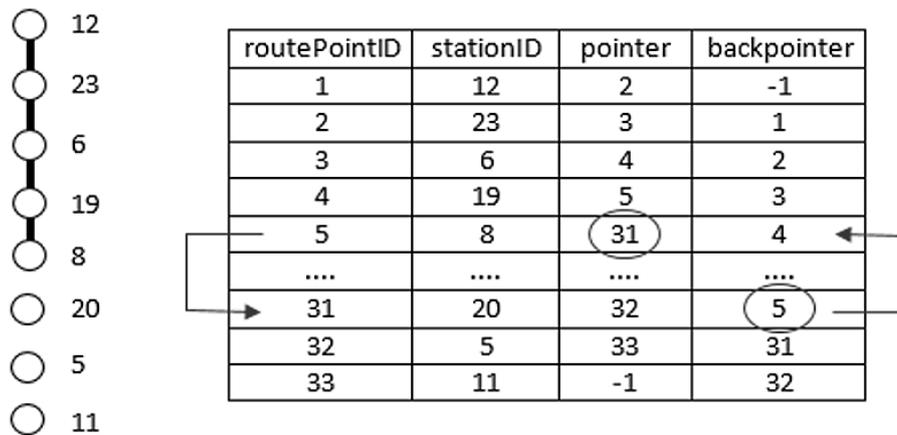
We will use the example of the Northern Line, as shown below. A sequence of stations from Edgware to Brent Cross is first entered, then a further section of line from Brent Cross to Belsize Park is added:



The first group of stations will be stored as a linked list, for example:



The additional sequence of stations will be attached by linking the pointer and backpointer values for the routePoint groups. There may be other records stored in the table between these blocks:



Open the **updateConnections.php** file and add the block of program code shown below. This begins by calling a function **checkConnections( )** which will determine whether the new sequence begins with the same station as the final station of a previously entered sequence. If so, a function **linkAfter( )** will add the new sequence to the end of the existing sequence with the pointers connected as in the example above.

```

echo"<p>StationID links:";
for ($i=0;$i<$linkCount;$i++)
{
    echo"<br>".$linkArray[$i];
}

$routePointCount=RoutePoint::loadPoints();
$result = checkConnections($lineID,$linkArray,$linkCount,$routePointCount);
$startRoutePointID=$result[0];
$finishRoutePointID=$result[1];
if (($startRoutePointID>0)&&($finishRoutePointID==0))
    linkAfter($linkCount,$linkArray,$startRoutePointID,$lineID);
else
if (($startRoutePointID==0)&&($finishRoutePointID==0))
    addLinks($lineID,$linkArray);
echo"<form method=post action='addRoutes.php'>";
echo"<p><input type=submit value='continue'>";
echo"</form>";
    
```

Go to the end of the **updateConnections.php** file and add the **checkConnections( )** function shown below.

```

function checkConnections($lineID,$linkArray,$linkCount,$routePointCount)
{
    $startID =$linkArray[0];
    $finishID =$linkArray[$linkCount-2];
    $startRouteID = 0;
    $finishRouteID = 0;
    for ($j=1;$j<=$routePointCount;$j++)
    {
        if(RoutePoint::$pointObj[$j]->lineID == $lineID)
        {
            if((RoutePoint::$pointObj[$j]->pointer == -1)||
                (RoutePoint::$pointObj[$j]->backpointer == -1))
            {
                if (RoutePoint::$pointObj[$j]->stationID == $startID)
                    $startRouteID =RoutePoint::$pointObj[$j]->routePointID;
                if (RoutePoint::$pointObj[$j]->stationID == $finishID)
                    $finishRouteID =RoutePoint::$pointObj[$j]->routePointID;
            }
        }
    }
    return array($startRouteID,$finishRouteID);
}

```

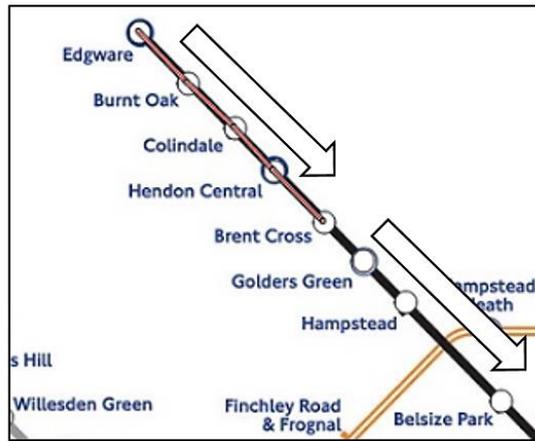
Continuing to work at the end of the **updateConnections.php** file, add the **linkAfter( )** function:

```

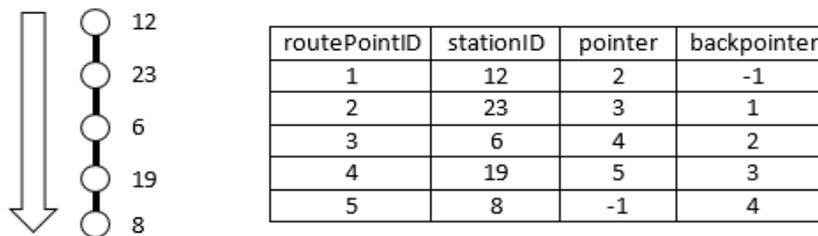
function linkAfter($linkCount,$linkArray,$startRouteID,$lineID)
{
    echo"<br>Existing segment goes first, then new addition";
    $pointer = RoutePoint::getPointerFromRoutePoint($startRouteID);
    if ($pointer == '-1')
        echo"<br>current sequence correct";
    else
    {
        echo"<br>current sequence must be reversed";
        RoutePoint::reverseSequence($startRouteID);
    }
    $linkCount= count($linkArray) - 1;
    $previous = $startRouteID;
    for ($i=1;$i<$linkCount;$i++)
    {
        $stationID=$linkArray[$i];
        $pointer=-1;
        $backpointer=$previous;
        $position=0;
        $routePointID = RoutePoint::addRoutePoint($lineID,$stationID,
            $pointer,$backpointer,$position);
        RoutePoint::updatePointer($previous,$routePointID);
        $previous=$routePointID;
    }
}

```

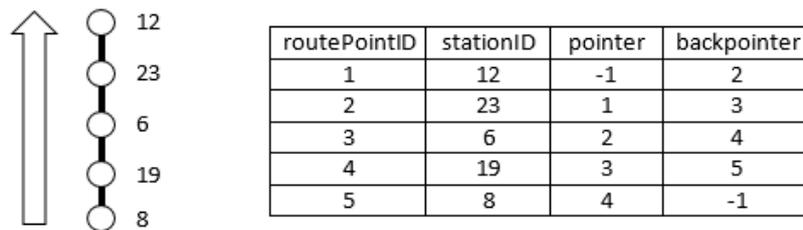
The function displays comments on screen, to allow us to check that it is operating correctly. A requirement for joining the new sequence of stations to the existing sequence is that both must run in the same direction along the route.



If the existing station sequence is found to run in the opposite direction, this can easily be reversed by swapping the **pointer** and **backpointer** values of the linked list. For example:



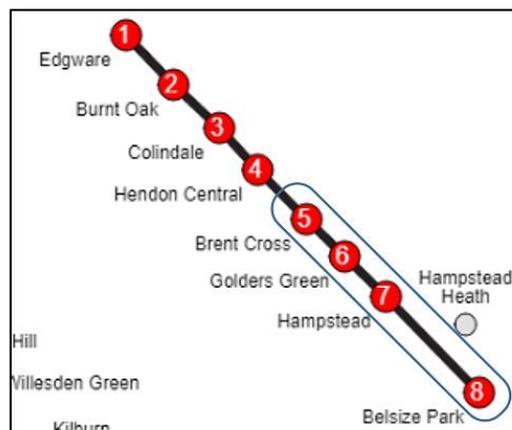
becomes:



Save the **updateConnections.php** file and copy it to the server.

Open the **RoutePoint.php** file and add the three methods shown on the next page. These carry out the exchange of the pointers and backpointers if required. Save the file and copy it to the server.

Run the website. Enter stations and a section of underground line, for example: the Northern line stations from Edgware to Brent Cross, numbered 1 – 5 below. Return to the 'Add link to route' option and select a further group of stations, beginning at the last point of the previous group – Brent Cross. After the new stations are entered, check that a continuous line appears on the 'View user map' page. Select the 'Check linkages' option, and make sure that all stations are linked in the correct number sequence.



```

public static function getPointerFromRoutePoint($routePoint)
{
    $pointer=0;
    $pointCount = RoutePoint::loadPoints();
    for($i=1;$i<=$pointCount;$i++)
    {
        if (RoutePoint::$pointObj[$i]->routePointID == $routePoint)
        {
            $pointer = RoutePoint::$pointObj[$i]->pointer;
        }
    }
    return $pointer;
}

public static function reverseSequence($startRouteID)
{
    $pointCount = RoutePoint::loadPoints();
    $current = $startRouteID;
    while($current>-1)
    {
        for($i=1;$i<=$pointCount;$i++)
        {
            if (RoutePoint::$pointObj[$i]->routePointID == $current)
            {
                $pointer = RoutePoint::$pointObj[$i]->pointer;
                $backpointer = RoutePoint::$pointObj[$i]->backpointer;
                RoutePoint::updatePointer($current,$backpointer);
                RoutePoint::updateBackpointer($current,$pointer);
                $current = $pointer;
                break;
            }
        }
    }
}

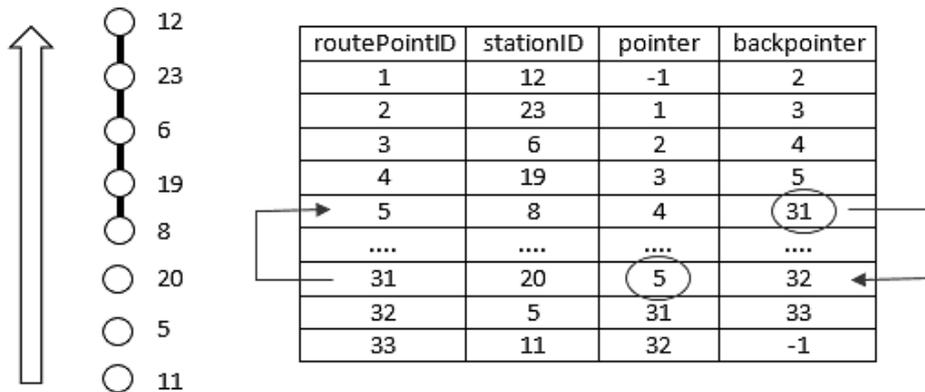
public static function updateBackpointer($IDwanted,$backpointer)
{
    include ('user.inc');
    $conn = new mysqli(localhost, $username, $password, $database);
    if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
    $query="UPDATE routePoint SET backpointer='$backpointer' WHERE
                                                routePointID='$IDwanted'";
    $result=mysqli_query($conn, $query);
    mysqli_close($conn);
}
}
?>

```

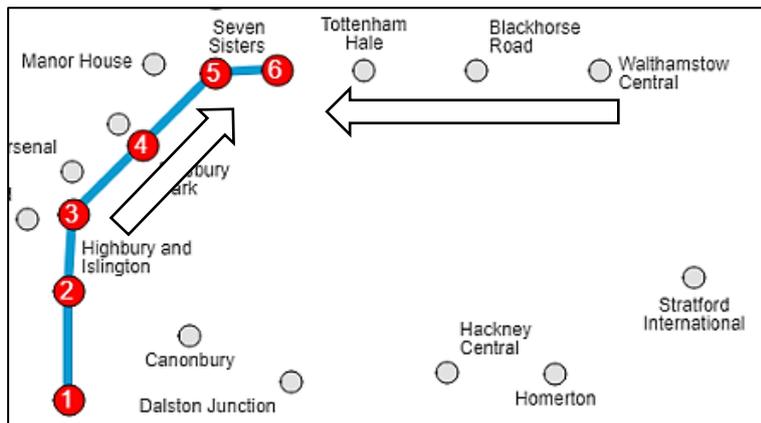
The next situation we must consider is the user adding a new sequence of stations *before* an existing section of route, for example:



The combined linked list will begin with the new section from Walthamstow to Seven Sisters, where it connects to an existing station sequence. Pointer and backpointer values will be adjusted to connect the two groups of route points.



It is essential that the new and existing linked list sequences run in the same direction along the underground line. In this example, the existing route points from Highbury to Seven Sisters would need to be reversed before the new linked section from Walthamstow is added.



Go to **updateConnections.php** and add another **else...** condition as shown. Save the file.

```

$startRoutePointID=$result[0];
$finishRoutePointID=$result[1];
if (($startRoutePointID>0)&&($finishRoutePointID==0))
    linkAfter($linkCount,$linkArray,$startRoutePointID,$lineID);
else
    linkBefore($linkCount,$linkArray,$finishRoutePointID,$lineID);
else
    if (($startRoutePointID==0)&&($finishRoutePointID>0))
        addLinks($lineID,$linkArray);
    echo"<form method=post action='addRoutes.php'>";
    
```

Go now to the **RoutePoint.php** class file and add a **reverseBacksequence( )** method. This method can work backwards from the end of an existing linked list by following backpointers. The pointer and backpointer values are swapped as it moves along the list, so that the final sequence is reversed.

```

public static function reverseBacksequence($finishRouteID)
{
    $pointCount = RoutePoint::loadPoints();
    $current = $finishRouteID;
    while($current > -1)
    {
        for($i=1;$i<=$pointCount;$i++)
        {
            if (RoutePoint::$pointObj[$i]->routePointID == $current)
            {
                $pointer = RoutePoint::$pointObj[$i]->pointer;
                $backpointer = RoutePoint::$pointObj[$i]->backpointer;
                RoutePoint::updatePointer($current,$backpointer);
                RoutePoint::updateBackpointer($current,$pointer);
                $current = $backpointer;
                break;
            }
        }
    }
}
?>

```

Save **RoutePoint.php** and copy it to the server.

Return to the **updateConnections.php** file and add the **linkBefore( )** method shown below.

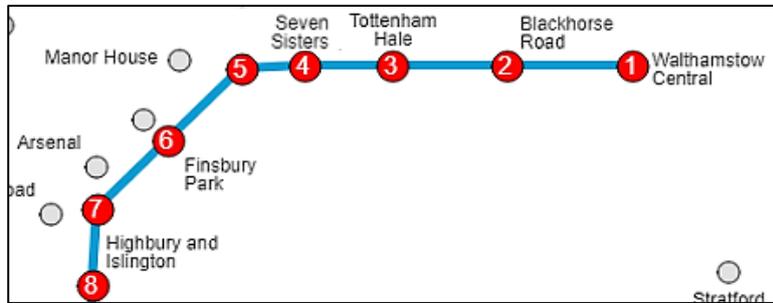
```

function linkBefore($linkCount,$linkArray,$finishRouteID,$lineID)
{
    echo"<br>New addition goes first, then existing segment";
    $pointer = RoutePoint::getPointerFromRoutePoint($finishRouteID);
    if ($pointer == '-1')
    {
        echo"<br>current sequence must be reversed";
        RoutePoint::reverseBacksequence($finishRouteID);
    }
    else
        echo"<br>current sequence correct";
    $linkCount= count($linkArray) - 1;
    for ($i=0;$i<$linkCount-1;$i++)
    {
        $stationID=$linkArray[$i];
        $pointer=-1;
        $backpointer=-1;
        if ($i>0)
            $backpointer=$previous;
        $position=0;
        $routePointID = RoutePoint::addRoutePoint($lineID,$stationID,
            $pointer,$backpointer,$position);
        RoutePoint::updatePointer($previous,$routePointID);
        $previous=$routePointID;
    }
    RoutePoint::updatePointer($previous,$finishRouteID);
    RoutePoint::updateBackpointer($finishRouteID,$previous);
}

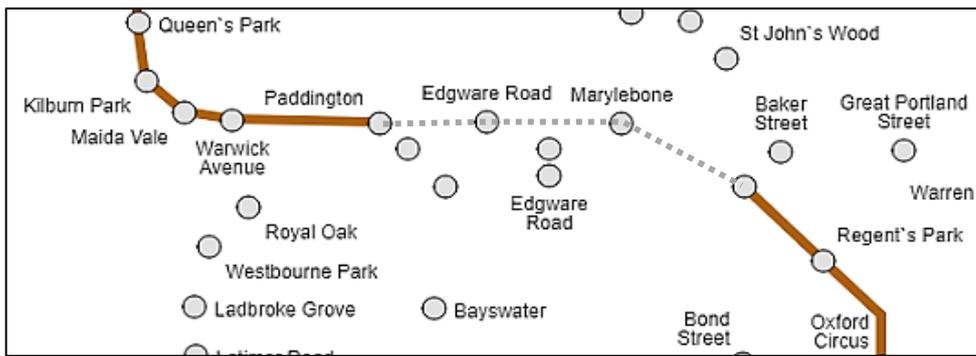
```

Save **updateConnections.php** and copy it to the server. Run the website and enter a section of the Victoria line, starting at Highbury & Islington, and ending at Seven Sisters. Please note when entering station names that the '&' symbol is a control character and can cause a problem. The station name should be entered as 'Highbury and Islington' with & replaced by 'and'.

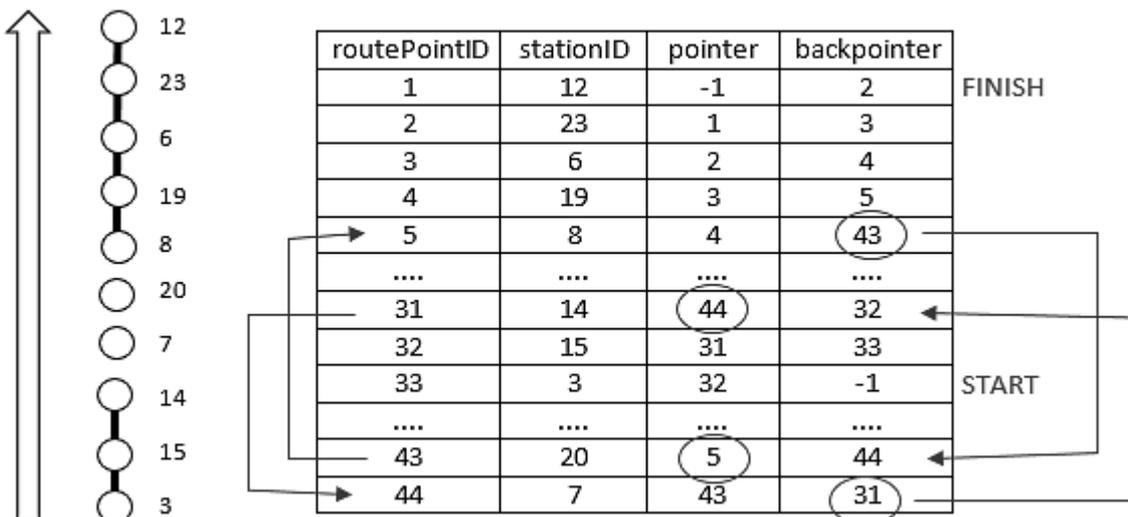
Return to the 'Add link to route' page and insert a station sequence from Walthamstow to Seven Sisters. Go to the 'View user map' page and use the 'Check linkages' option to check that the stations along the Victoria line are correctly connected as a single linked list.



One further possibility when entering a route is that two existing station sequences need to be connected, as in this example for the Bakerloo line:



The combined linked list will be created by adding route points for the new intermediate stations **Edgware Road** and **Marylebone**, then adjusting the pointers and backpointers to connect with the two existing route sections.



Begin by going to the **updateConnections.php** and add another **else...** condition as shown below.

```

if (($startRoutePointID>0)&&($finishRoutePointID==0))
    linkAfter($linkCount,$linkArray,$startRoutePointID,$lineID);
else
if (($startRoutePointID==0)&&($finishRoutePointID>0))
    linkBefore($linkCount,$linkArray,$finishRoutePointID,$lineID);
else
if (($startRoutePointID>0)&&($finishRoutePointID>0))
    linkBetween($linkCount,$linkArray,$startRoutePointID,$finishRoutePointID,$lineID);
else
if (($startRoutePointID==0)&&($finishRoutePointID==0))

```

Move to the end of the **updateConnections.php** file and add the **linkBetween( )** function. This will add any new intermediate route points and adjust the pointer and backpointer values to connect with the existing route sections.

```

function linkBetween($linkCount,$linkArray,$startRouteID,$finishRouteID,$lineID)
{
    echo"<br>New addition goes between two existing segments";
    $pointer = RoutePoint::getPointerFromRoutePoint($startRouteID);
    if ($pointer == '-1')
        echo"<br>current sequence before correct";
    else
    {
        echo"<br>current sequence before must be reversed";
        RoutePoint::reverseSequence($startRouteID);
    }
    $pointer = RoutePoint::getPointerFromRoutePoint($finishRouteID);
    if ($pointer == '-1')
    {
        echo"<br>current sequence after must be reversed";
        RoutePoint::reverseBacksequence($finishRouteID);
    }
    else
        echo"<br>current sequence after correct";
    $linkCount= count($linkArray) - 1;
    $previous = $startRouteID;
    if ($linkCount>2)
    {
        for ($i=1;$i<($linkCount-1);$i++)
        {
            $stationID=$linkArray[$i];
            $pointer=-1;
            $backpointer=$previous;
            $position=0;
            $routePointID = RoutePoint::addRoutePoint($lineID,$stationID,
                $pointer,$backpointer,$position);
            RoutePoint::updatePointer($previous,$routePointID);
            $previous=$routePointID;
        }
    }
    RoutePoint::updatePointer($previous,$finishRouteID);
    RoutePoint::updateBackpointer($finishRouteID,$previous);
}

```

Save **updateConnections.php** and copy it to the server.

Run the website and check that two sections of a route can be entered separately and then linked. Use the test function to check that the sequence of stations is shown correctly. Numbering of route points should run continuously through the connecting section.

Before leaving the 'Add link to route' page, it would be useful to add an option to **delete** links from the map. This may be necessary if an error is made during data entry.

Open the **addRoutes.php** file. Add lines of code to the **setup( )** function to produce a 'remove link' button.

```

        button = createButton('clear');
        button.position(1200, 200);
        button.mousePressed(clearLinks);

        button = createButton('remove link');
        button.position(1100, 240);
        button.mousePressed(removeLink);
    }
    function draw()

```

Add a function **removeLink( )** at the end of the **<script>** block. This function will be activated when the button is clicked, and will load the **updateConnections.php** file. The station IDs indicating the link to be deleted will be transferred as the variable **linkString**. Save the **addRoutes.php** file and copy it to the server.

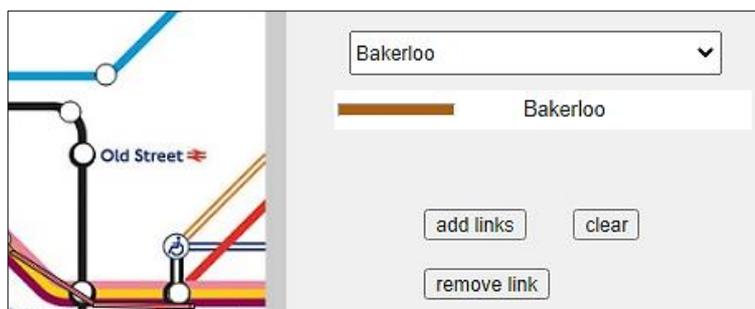
```

function removeLink()
{
    result = confirm('Remove this link for the '+lineWanted+' line');
    if (result==true)
    {
        window.location = "updateConnections.php?remove=YES&lineName="+
            sel.value()+"&linkString="+linkString;
    }
}

</script>
</body>
</html>

```

Save the **addRoutes.php** file and copy it to the server. Run the 'Add link to route' page and check that the 'remove link' button appears to the right of the map.



Go now to the **updateConnections.php** file and add lines of program to the PHP section near the start of the file as shown. A bracket pair for the **else...** condition must be created around a block of program code, as shown below.

```

for ($i=0;$i<$linkCount;$i++)
{
    echo"<br>".$linkArray[$i];
}
$routePointCount=RoutePoint::loadPoints();

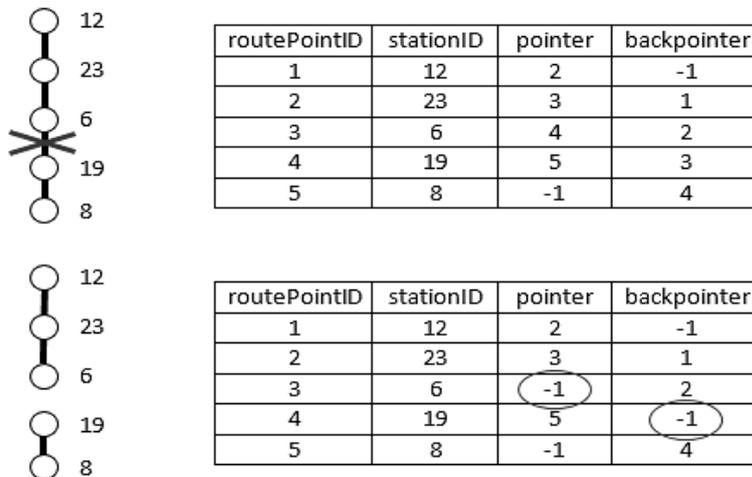
$remove=$_REQUEST['remove'];
if ($remove=='YES')
{
    removeLink($linkArray,$lineID,$linkCount,$routePointCount);
}
else
{
    $result = checkConnections($lineID,$linkArray,$linkCount,$routePointCount);
    .....

    else
    if (($startRoutePointID==0)&&($finishRoutePointID==0))
        addLinks($lineID,$linkArray);
}

echo"<form method=post action='addRoutes.php'>";
echo"<p><input type=submit value='continue'>";

```

A function **removeLink()** will remove the connection between two stations, splitting a single linked list into two separate lists. This is done by resetting pointer and backpointer values to -1. For example:



Open the **RoutePoint.php** class file and add two small methods which will be needed during the deletion of a link, as shown in the boxes below.

```

public static function getBackpointerFromRoutePoint($routePoint)
{
    $pointer=0;
    $pointCount = RoutePoint::loadPoints();
    for($i=1;$i<=$pointCount;$i++)
    {
        if (RoutePoint::$pointObj[$i]->routePointID == $routePoint)
            $pointer = RoutePoint::$pointObj[$i]->backpointer;
    }
    return $pointer;
}
}
?>

```

```

public static function deleteRoutePoint($routePointID)
{
    include('user.inc');
    $conn = new mysqli(localhost, $username, $password, $database);
    if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
    $query = "DELETE FROM routePoint WHERE routePointID = '$routePointID'";
    $result=mysqli_query($conn, $query);
    mysqli_close($conn);
}
}
?>

```

Save the **RoutePoint.php** file and copy it to the server.

Return to **updateConnections.php** and add a function **checkConnections2( )** at the end of the file.

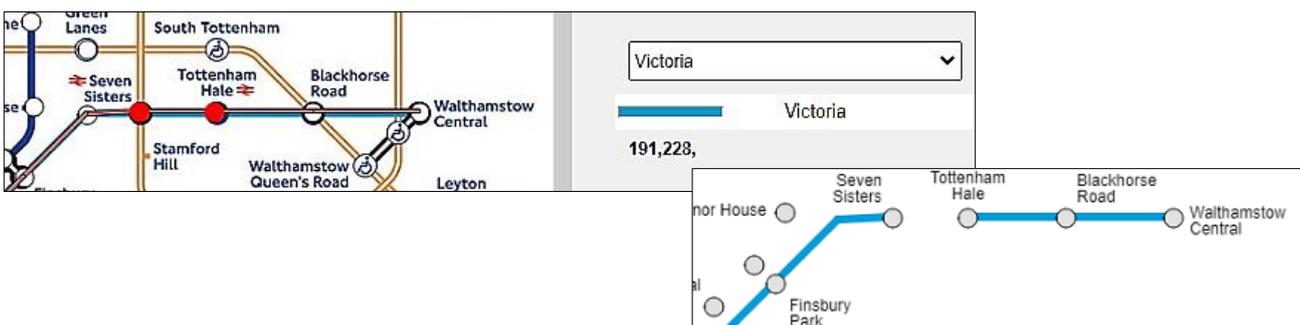
```

function checkConnections2($lineID,$linkArray,$linkCount,$routePointCount)
{
    $startID =$linkArray[0];
    $finishID =$linkArray[1];
    $startRouteID = 0;
    $finishRouteID = 0;
    for ($j=1;$j<=$routePointCount;$j++)
    {
        if(RoutePoint::$pointObj[$j]->lineID == $lineID)
        {
            if(RoutePoint::$pointObj[$j]->stationID == $startID)
                $startRouteID=RoutePoint::$pointObj[$j]->routePointID;
            if(RoutePoint::$pointObj[$j]->stationID == $finishID)
                $finishRouteID=RoutePoint::$pointObj[$j]->routePointID;
        }
    }
    return array($startRouteID,$finishRouteID);
}
?>

```

Finally, add the **removeLink( )** function at the end of the **updateConnections.php** file as shown on the next page. The function handles the additional possibility of a link being removed from the beginning or end of a station sequence. In this case, the linked list is simply shortened by removing the end record, rather than splitting the list into two.

Save the **updateConnections.php** file and copy it to the server. Run the website and go to the 'Add link to route' page. Links are removed one at a time. Select two adjacent stations on a route section as in the example below. Select the corresponding underground line from the drop-down list, then click the 'remove link' button. Check that the selected link has now been removed from the route.



```

function removeLink($linkArray,$lineID,$linkCount,$routePointCount)
{
    $startStationID=$linkArray[0];
    $finishStationID=$linkArray[1];
    $result = checkConnections2($lineID,$linkArray,$linkCount,$routePointCount);
    $startRouteID=$result[0];
    $finishRouteID=$result[1];
    $pointerS = RoutePoint::getPointerFromRoutePoint($startRouteID);
    $backpointerS = RoutePoint::getBackpointerFromRoutePoint($startRouteID);
    $pointerF = RoutePoint::getPointerFromRoutePoint($finishRouteID);
    $backpointerF = RoutePoint::getBackpointerFromRoutePoint($finishRouteID);
    if($backpointerS==$finishRouteID)
    {
        if ($pointerS== -1)
            RoutePoint::deleteRoutePoint($startRouteID);
        else
            RoutePoint::updateBackpointer($startRouteID, '-1');
        if ($backpointerF== -1)
            RoutePoint::deleteRoutePoint($finishRouteID);
        else
            RoutePoint::updatePointer($finishRouteID, '-1');
    }
    else
    if($pointerS==$finishRouteID)
    {
        if ($backpointerS== -1)
            RoutePoint::deleteRoutePoint($startRouteID);
        else
            RoutePoint::updatePointer($startRouteID, '-1');
        if ($pointerF== -1)
            RoutePoint::deleteRoutePoint($finishRouteID);
        else
            RoutePoint::updateBackpointer($finishRouteID, '-1');
    }
}
}

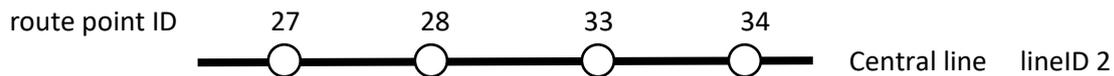
```

?>

This completes the entry procedure for simple underground lines which run along an unshared route. We must, however, allow for the entry of parallel underground lines running between pairs of stations. For example, the **Bakerloo**, **Circle** and **Hammersmith and City** lines share the route between Baker Street and Moorgate:



A strategy is to create a table of all existing links between route points along with the lineID of the underground line forming the link, as in the example below. When a new route segment is to be added, the number of underground lines already passing between pairs of stations can be counted.



route link	from	to	lineID
1	27	28	2
2	28	33	2
3	33	34	2

Go to the end of the **updateConnections.php** file and add a function to create the table as a two-dimensional array with the name **\$routeLink**.

```
function makeLinkArray($lineCount,$routePointCount)
{
    $oldStationID=0;
    $linkCount=0;
    $routeLink = array();
    $routeLink[] = array();
    for ($i=1;$i<=$lineCount;$i++)
    {
        $currentLineID=Line::$lineObj[$i]->lineID;
        for ($j=1;$j<=$routePointCount;$j++)
        {
            if ((RoutePoint::$pointObj[$j]->backpointer==-1)&&
                (RoutePoint::$pointObj[$j]->lineID==$currentLineID))
            {
                $currentPointer=RoutePoint::$pointObj[$j]->pointer;
                $stationIDwanted=RoutePoint::$pointObj[$j]->stationID;
                $oldStationID=$stationIDwanted;
                $oldRoutePoint=$j;
                $finished=false;
                while($finished==false)
                {
                    for ($n=1;$n<=$routePointCount;$n++)
                    {
                        if (RoutePoint::$pointObj[$n]->routePointID==$currentPointer)
                            $pos=$n;
                    }
                    $newPointer=RoutePoint::$pointObj[$pos]->pointer;
                    $stationIDwanted=RoutePoint::$pointObj[$pos]->stationID;
                    $newRoutePoint=$pos;
                    if ($currentPointer<0)
                        $finished=true;
                    else
                    {
                        $linkCount++;
                        $routeLink[$linkCount][0]= $oldStationID;
                        $routeLink[$linkCount][1]= $stationIDwanted;
                        $routeLink[$linkCount][2]= $currentLineID;
                        $currentPointer=$newPointer;
                        $oldStationID=$stationIDwanted;
                        $oldRoutePoint=$newRoutePoint;
                    }
                }
            }
        }
    }
    return $routeLink;
}
```

The function checks the linked lists for each of the existing route sequences, and adds each pair of route points as a new row in the array, along with the ID value for the underground line connecting them.

Continuing to work in the **updateConnections.php** file, add the function **checkMultiple( )** as shown.

```
function checkMultiple($previousStationID, $stationID,
                      $nextStationID, $routeLink, $lineID)
{
    $position = 0;
    $arrayCount=0;
    $lineArray = array();
    $a = $stationID;
    for ($h=1;$h<count($routeLink);$h++)
    {
        $b = $nextStationID;
        if ($b<1)
            $b = $previousStationID;
        if (((($routeLink[$h][0]==$a)&&($routeLink[$h][1]==$b))||
            (($routeLink[$h][0]==$b)&&($routeLink[$h][1]==$a))))
        {
            $currentLine=$routeLink[$h][2];
            $found=false;
            for ($p=0;$p<$arrayCount;$p++)
            {
                if ($lineArray[$p]==$currentLine)
                    $found=true;
            }
            if ($found==false)
            {
                $lineArray[$arrayCount]=$currentLine;
                $arrayCount++;
            }
        }
        $position=count($lineArray);
    }
    return $position;
}
```

The function will be called before each new underground line link between pairs of stations is added to the map. The function accesses the **\$routeLink** array and returns a variable **\$position**, which is the number of existing links between the pair of station points. For example:



The **\$position** value will then be stored in the routePoint record, along with the lineID for the underground line to be added. This value can then be used to offset the new line on the map if necessary, so that any existing lines are not obscured.

Add lines of program code near the start of **updateConnections.php** to create the **\$routeLink** array, extend the array to two dimensions, then call the **makeLinkArray( )** function.

```

for ($i=0;$i<$linkCount;$i++)
{
    echo"<br>".$linkArray[$i];
}
$routePointCount=RoutePoint::loadPoints();

$routePointCount=RoutePoint::loadPoints();
$lineCount=Line::loadLines();
$routeLink = array();
$routeLink[] = array();
$routeLink = makeLinkArray($lineCount,$routePointCount);

$remove=$_REQUEST['remove'];
if ($remove=='YES')

```

Let us consider the case of entering a new line segment which is not connected to any existing section of the same underground line. Locate the **addLinks()** function in the **updateConnections.php** file. Replace this with a new version of the function as shown below:

```

function addLinks($lineID,$linkArray,$routeLink)
{
    $linkCount= count($linkArray);
    for ($i=0;$i<($linkCount-1);$i++)
    {
        $previousStationID=0;
        if ($i>0)
            $previousStationID=$linkArray[$i-1];
        $stationID=$linkArray[$i];
        $nextStationID=$linkArray[$i+1];
        $position = checkMultiple($previousStationID,$stationID,
                                $nextStationID,$routeLink,$lineID);

        $pointer=-1;
        $backpointer=-1;
        if ($i>0)
            $backpointer=$previous;
        $routePointID = RoutePoint::addRoutePoint($lineID,$stationID,
                                                  $pointer,$backpointer,$position);

        if ($i>0)
            RoutePoint::updatePointer($previous,$routePointID);
        $previous=$routePointID;
    }
}

```

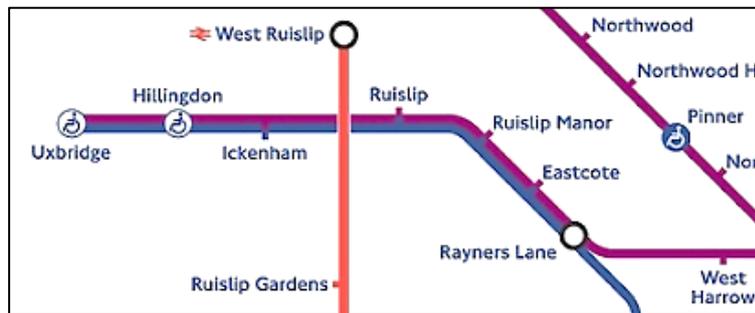
Go to the section near the start of the **updateConnections.php** file and make a change to the line where the **addLinks()** function is called. Add the **\$routeLink** array as another input parameter.

```

else
if (($startRoutePointID>0)&&($finishRoutePointID>0))
    linkBetween($linkCount,$linkArray,$startRoutePointID,$finishRoutePointID,$lineID);
else
if (($startRoutePointID==0)&&($finishRoutePointID==0))
    addLinks($lineID,$linkArray,$routeLink);
}
echo"<form method=post action='addRoutes.php'>";
echo"<p><input type=submit value='continue'>";
echo"</form>";

```

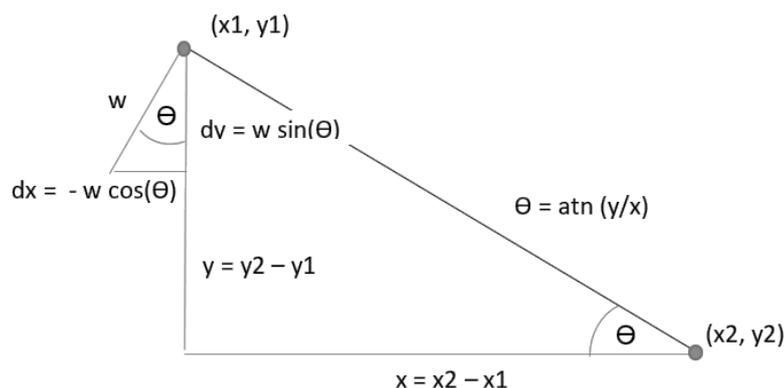
Save **updateConnections.php** and copy it to the server. Run the website and go to the 'Add link to route' page. Select a point on the map where two different underground lines run alongside each other, such as the Piccadilly and Metropolitan lines between Uxbridge and Rayners Lane:



Enter one of the lines, followed by the other. Go to the PHP MyAdmin web page and examine the routePoint table. Points along the first line entered should have a position value of 0, while the points along the second line should have a position value of 1.

routePointID	lineID	stationID	pointer	backpointer	position
399	15	406	400	398	0
400	15	404	401	399	0
401	15	405	402	400	0
402	15	382	-1	401	0
403	10	133	404	-1	1
404	10	404	405	403	1
405	10	405	406	404	1
406	10	382	407	405	1

When the second underground line is added to the map display, it should be offset to avoid being drawn on top of the first line. The necessary offset can be calculated using trigonometry. Suppose that a line connecting two stations is to be drawn from the point  $(x_1, y_1)$  to the point  $(x_2, y_2)$ . The angle  $\theta$  which this line makes with the horizontal can be found using an **arc tangent** function, as in the diagram below.



For the new underground line to be shown running alongside an existing line, a small offset at  $90^\circ$  to the route should be applied. The offset will be at the same angle  $\theta$ , but this time in relation to the vertical. For an offset distance of  $w$ , the horizontal  $dx$  and vertical  $dy$  components can be found using **cosine** and **sine** functions.

Open the **mapFunctions.php** file and add two small functions to calculate the  $x$  and  $y$  offset values as shown below.

```

function xOffset(x1,y1,x2,y2)
{
  x=int(x2-x1);
  y=int(y2-y1);
  a = Math.atan(y/x);
  dx= 6 * Math.cos(HALF_PI+a);
  return dx;
}

function yOffset(x1,y1,x2,y2)
{
  x=x2-x1;
  y=y2-y1;
  a = Math.atan(y/x);
  dy= 6 * Math.sin(HALF_PI+a);
  return dy;
}

```

Update the **plotLines2()** function by adding or replacing the lines of program code indicated below and on the following page.

```

function plotLines2()
{
  var x1=0;
  var y1=0;
  var x2=0;
  var y2=0;

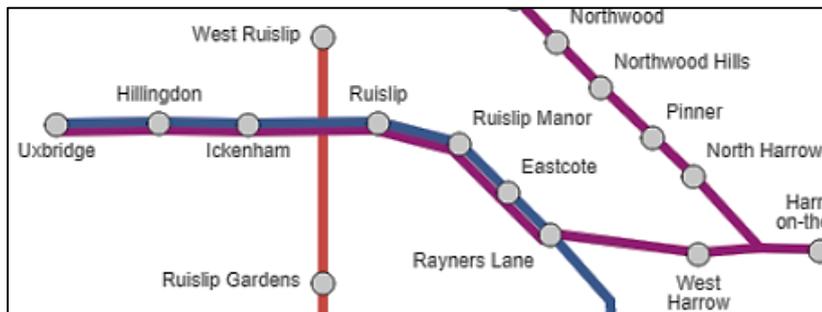
  for (i=1;i<=lineCount;i++)
  {
    stroke(255);
    fill(0);
    currentLineID=lineObj[i].lineID;
    colourCode=lineObj[i].colourCode;
    solid=lineObj[i].solid;
    let c = split(colourCode, ',');
    r=int(c[0]); g=int(c[1]); b=int(c[2]);
    fill(r,g,b);
    stroke(r,g,b);
    for (j=1;j<=pointCount;j++)
    {
      if ((pointObj[j].backpointer==-1)&&(pointObj[j].lineID==currentLineID))
      {
        currentPointer=pointObj[j].pointer;
        stationIDwanted=pointObj[j].stationID;
        positionWanted=pointObj[j].position;

        finished=false;
        oldX=getXfromID(stationIDwanted);
        oldY=getYfromID(stationIDwanted);
        oldX = int(oldX-transH);
        oldY = int(oldY-transV);
        finished=false;
        count=0;
        while(finished==false)
        {
          for (n=1;n<=pointCount;n++)
          {
            if (pointObj[n].routePointID==currentPointer)

```



Save the **mapFunctions.php** file and copy it to the server. Run the web page, selecting the 'View user map' option. The two underground lines should be shown running alongside one another without overlapping.



We can now update the functions which allow sections of line to be input before, after or between existing sections of the same underground line. This will allow the user more flexibility when entering long sections of track which run alongside other lines for parts of their route.

Return to the **updateConnections.php** file and change other function calls to include the **\$routeLink** array as shown:

```

$startRoutePointID=$result[0];
$finishRoutePointID=$result[1];
if (($startRoutePointID>0)&&($finishRoutePointID==0))
    linkAfter($linkCount,$linkArray,$startRoutePointID,$lineID,$routeLink);
else
if (($startRoutePointID==0)&&($finishRoutePointID>0))
    linkBefore($linkCount,$linkArray,$finishRoutePointID,$lineID,$routeLink);
else
if (($startRoutePointID>0)&&($finishRoutePointID>0))
    linkBetween($linkCount,$linkArray,$startRoutePointID,
                $finishRoutePointID,$lineID,$routeLink);
else
if (($startRoutePointID==0)&&($finishRoutePointID==0))
    addLinks($lineID,$linkArray,$routeLink);
}
echo"<form method=post action='addRoutes.php'>";
echo"<p><input type=submit value='continue'>";
echo"</form>";
    
```

Locate the **linkAfter( )** function. Alter the heading line.

```

function linkAfter($linkCount,$linkArray,$startRouteID,$lineID,$routeLink)
{
    echo"<br>Existing segment goes first, then new addition";
    $pointer = RoutePoint::getPointerFromRoutePoint($startRouteID);
    if ($pointer == '-1')
        echo"<br>current sequence correct";
}
    
```

Add or alter the lines of the **linkAfter( )** function indicated below.

```

$linkCount= count($linkArray) - 1;
$previous = $startRouteID;
for ($i=1;$i<$linkCount;$i++)
{
    $previousStationID=0;
    if ($i>1)
        $previousStationID=$linkArray[$i-1];

    $stationID=$linkArray[$i];

    $nextStationID=$linkArray[$i+1];

    $pointer=-1;
    $backpointer=$previous;

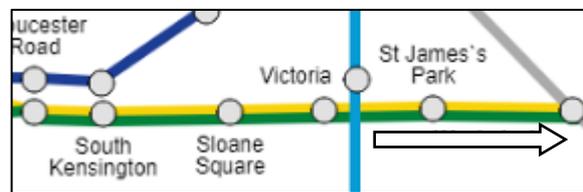
    $position = checkMultiple($previousStationID,$stationID,
                             $nextStationID,$routeLink,$lineID);

    $routePointID = RoutePoint::addRoutePoint($lineID,$stationID,
                                              $pointer,$backpointer,$position);

    RoutePoint::updatePointer($previous,$routePointID);
    $previous=$routePointID;
}
}

```

Save the **updateConnections.php** file and copy it to the server. Run the web page, selecting the 'Add link to route' option. Choose a section of the map where two or three underground lines are shown running alongside one another. Enter one line, followed by the first section of a second line. Check that the remaining section of line can then be entered and displayed correctly. For example:



Return to **updateConnections.php** and update the **linkBefore()** function in a similar way by adding or altering the lines indicated in the two boxes below:

```

function linkBefore($linkCount,$linkArray,$finishRouteID,$lineID,$routeLink)
{
    echo"<br>New addition goes first, then existing segment";
    $pointer = RoutePoint::getPointerFromRoutePoint($finishRouteID);

    $linkCount= count($linkArray) - 1;
    for ($i=0;$i<$linkCount-1;$i++)
    {
        $previousStationID=0;
        if ($i>0)
            $previousStationID=$linkArray[$i-1];

        $stationID=$linkArray[$i];

        $nextStationID=$linkArray[$i+1];

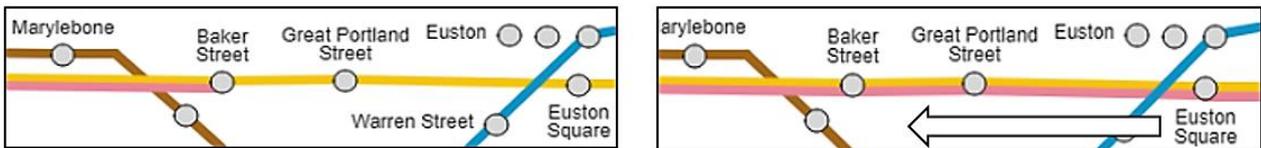
        $pointer=-1;
        $backpointer=-1;
    }
}

```

```

    if ($i>0)
        $backpointer=$previous;
    $position = checkMultiple($previousStationID,$stationID,
                             $nextStationID,$routeLink,$lineID);
    $routePointID = RoutePoint::addRoutePoint($lineID,$stationID,
                                               $pointer,$backpointer,$position);
    RoutePoint::updatePointer($previous,$routePointID);
    $previous=$routePointID;
}
RoutePoint::updatePointer($previous,$finishRouteID);
RoutePoint::updateBackpointer($finishRouteID,$previous);
}
    
```

Save the **updateConnections.php** file and copy it to the server. Run the web page, selecting the 'Add link to route' option. Select a point where multiple lines run in parallel. Enter one line, followed by a section of a second line. Check that another group of stations can be added before this, for example:



Finally, update the **linkBetween()** function by adding or altering program lines as indicated below:

```

function linkBetween($linkCount,$linkArray,$startRouteID,
                    $finishRouteID,$lineID,$routeLink)
{
    echo"<br>New addition goes between two existing segments";
    $pointer = RoutePoint::getPointerFromRoutePoint($startRouteID);

    $linkCount= count($linkArray) - 1;
    $previous = $startRouteID;
    if ($linkCount>2)
    {
        for ($i=1;$i<($linkCount-1);$i++)
        {
            $previousStationID=0;
            if ($i>1)
                $previousStationID=$linkArray[$i-1];

            $stationID=$linkArray[$i];

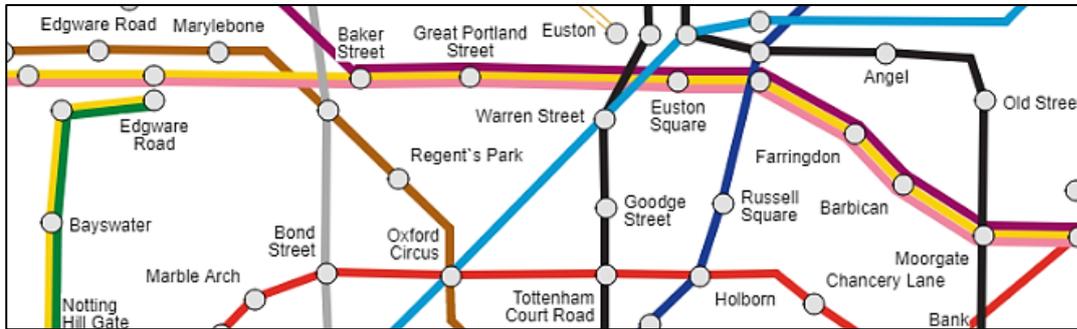
            $nextStationID=$linkArray[$i+1];

            $pointer=-1;
            $backpointer=$previous;

            $position = checkMultiple($previousStationID,$stationID,
                                     $nextStationID,$routeLink,$lineID);

            $routePointID = RoutePoint::addRoutePoint($lineID,$stationID,
                                                       $pointer,$backpointer,$position);
            RoutePoint::updatePointer($previous,$routePointID);
            $previous=$routePointID;
        }
    }
    RoutePoint::updatePointer($previous,$finishRouteID);
}
    
```

Save the **updateConnections.php** file and copy it to the server. Run the web page, selecting the 'Add link to route' option. Multiple lines can now be entered, with sections added and joined to form complete underground routes. You may wish to make backups of the **routePoint** table at intervals, using the **export** option in the database. This will allow the data to be restored if a problem occurs as you build up the map.



This completes the data entry for the London Underground route planning application. We can now move on to create the public web pages. Open a blank file and add the program code shown below. Save the file as **index.php** and copy it to the server.

```
<html>
<head>
  <title>London Underground route planning</title>
  <link rel="stylesheet" type="text/css" href="styleSheet.css" />
  <script src="p5.js"></script>
  <script src="p5.dom.js"></script>
</head>
<body>
  <br>
  <script type="text/javascript">
  function setup()
  {
    createCanvas(1000, 654);
  }

  function draw()
  {
    background(255);
  }
  </script>
</body>
</html>
```

Run the website. The **index.php** page will be loaded by default if no other file name is specified. A white rectangular window should appear on the page, ready for the addition of the underground map.

Open the **styleSheet.css** file and add formatting commands for a **lineTable2** division. Save the file and copy it to the server.

```
div.lineTable2 {
  position: absolute;
  top: 700px;
  left: 100px;
  width: 800px;
  background-color: white;
}
```

It will be convenient to display a key beneath the map listing the lines and showing their colour codes. Open the **Line.php** class file and add the **linelist2()** method shown below. Save the file and copy it to the server.

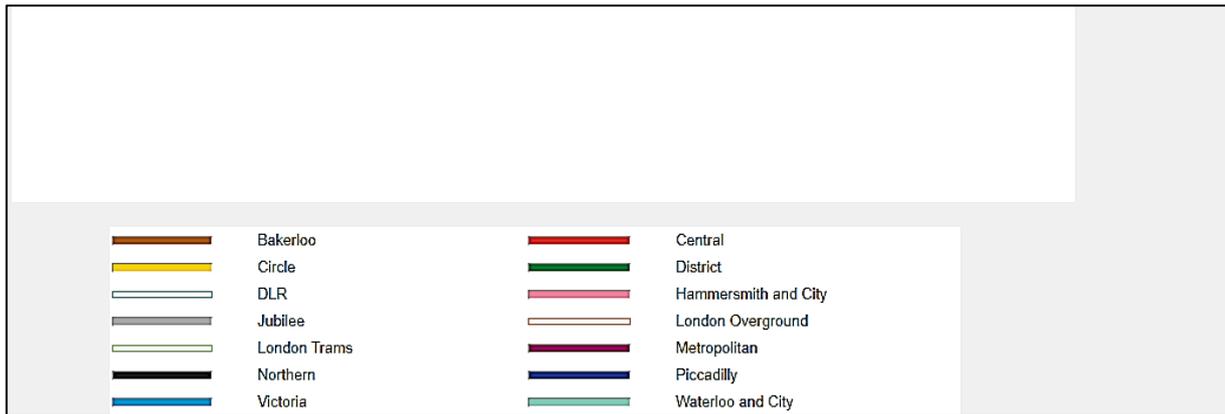
```
public static function linelist2($lineCount)
{
    echo"<div class='lineTable2'>";
    echo"<table border=0>";
    for ($i=1;$i<=$lineCount; $i++)
    {
        $lineName=Line::$lineObj[$i]->lineName;
        $colourCode=Line::$lineObj[$i]->colourCode;
        $solid=Line::$lineObj[$i]->solid;
        if ($i%2==1)
            echo" <tr height=5px >";
        echo" <td width=160px>";
        if ($solid==true)
        {
            echo"<hr size='8' style='background-color:rgb(".$colourCode.");'></td>";
        }
        else
        {
            echo"<hr style='height: 4px; border: 1px
                solid rgb(".$colourCode.");'></td>";
        }
        echo" <td width=60></td>";
        echo" <td width=260 style='font-size: 14px;'>".$lineName."</td>";
        echo" <td width=120></td>";
    }
    echo"</table>";
    echo"</div>";
}
```

Return to **index.php**. Add lines of PHP code at the start of the file to load and display the list of underground lines.

```
<?
    include('Line.php');
    $lineCount=Line::loadLines();
    Line::linelist2($lineCount);
?>

<html>
<head>
    <title>London Underground route planning</title>
```

Save **index.php** and copy it to the server. Run the web site and check that the key is displayed at the bottom of the page, as shown in the example below. It may be necessary to hold down the CTRL key whilst reloading the page, so that the style sheet is updated.



The next step is to display the route map in a scrolling window. Return to **index.php** and add lines of code to the PHP section at the start of the file.

```
<?
  include('Line.php');
  $lineCount=Line::loadLines();
  Line::linelist2($lineCount);

  $Hscroll=$_REQUEST['Hscroll'];
  $Vscroll=$_REQUEST['Vscroll'];
  include('mapFunctions.php');

?>
<html>
<head>
  <title>London Underground route planning</title>
```

Move down to the **<script>** block and add lines of code as shown below. The program makes use of functions which we wrote earlier to display the underground lines and station symbols, and to add horizontal and vertical scroll bars to the map window.

```
<script type="text/javascript">
  var VscrollPosition=300;
  var HscrollPosition=400;
  Hscroll = <? echo json_encode($Hscroll); ?>;
  Vscroll = <? echo json_encode($Vscroll); ?>;
  var Hscroll=false;
  var Vscroll=false;

  function setup()
  {
    if ((Hscroll>0)|| (Vscroll>0))
    {
      VscrollPosition=Vscroll;
      HscrollPosition=Hscroll;
    }

    createCanvas(1000, 654);
  }
}
```

Add lines of program code to the **draw()** method as shown below.

```

function draw()
{
    transV = map(VscrollPosition, 0, (height-14), 0, 1890-height);
    transH = map(HscrollPosition, 0, (width-14), 0, 2560-width);
    push();
    translate(-transH, -transV);

    background(255);

    pop();
    plotLines2();
    displayNames();
    displayStations(225,225,225,"NO");
    Hscrollbar(HscrollPosition);
    Vscrollbar(VscrollPosition);
    x=mouseX;
    y=mouseY;
    scrollMove();
}
</script>

```

It will be necessary to load the PHP objects representing underground lines, stations and route points, and convert these to equivalent JavaScript objects. Begin by inserting lines of PHP code near the beginning of the **index.php** file, as shown below.

```

    $Hscroll=$_REQUEST['Hscroll'];
    $Vscroll=$_REQUEST['Vscroll'];
    include('mapFunctions.php');

    include ('Stations.php');
    $stationCount=Stations::loadStations();
    include('RoutePoint.php');
    $pointCount=RoutePoint::loadPoints();

?>
<html>
<head>
    <title>London Underground route planning</title>

```

Go now to the **<script>** block and insert lines of code to convert data to JavaScript objects and variables by means of JSON encoding.

```

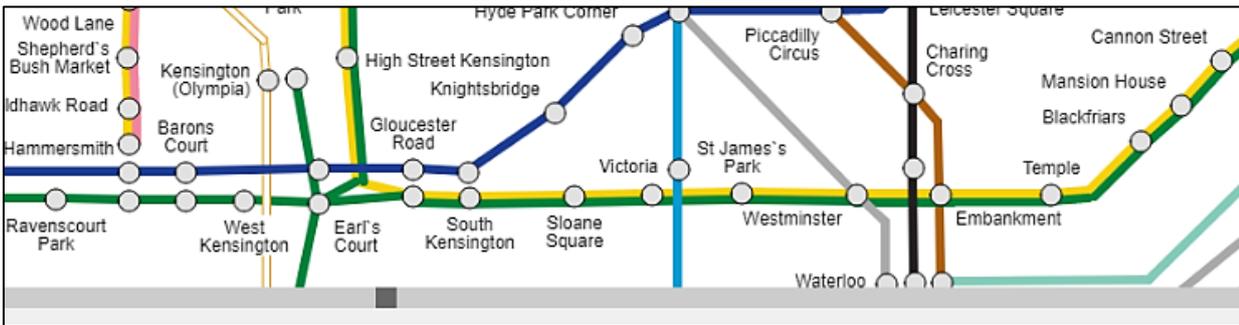
var HscrollPosition=400;
Hscroll = <? echo json_encode($Hscroll); ?>;
Vscroll = <? echo json_encode($Vscroll); ?>;
var Hscroll=false;
var Vscroll=false;

stationObj = <? echo json_encode(Stations::$stationObj); ?>;
stationCount = <? echo json_encode($stationCount); ?>;
stationList = <? echo json_encode($stationList); ?>;
listCount = <? echo json_encode($listCount); ?>;
lineObj = <? echo json_encode(Line::$lineObj); ?>;
lineCount = <? echo json_encode($lineCount); ?>;
pointObj = <? echo json_encode(RoutePoint::$pointObj); ?>;
pointCount = <? echo json_encode($pointCount); ?>;

function setup()
{

```

Save the **index.php** file and copy it to the server. Run the web page and check that the route map entered earlier is visible and can be scrolled in the screen window.

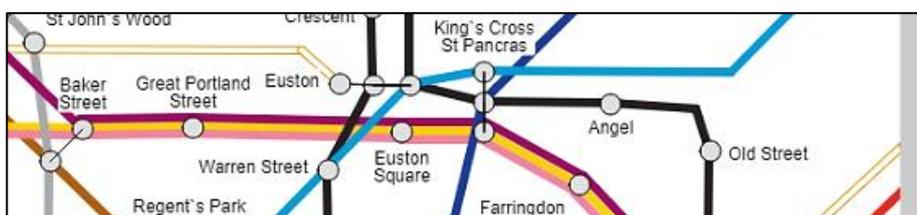


The next step is to add HTML components to the panel on the right of the map to produce a user interface for the route information system. In preparation for this, open the **Stations.php** class file and add a **loadStationList()** method as shown below. This will create an alphabetical list of station names which can be used in a drop-down selection box.

```
public static function loadStationList()
{
    include ('user.inc');
    $conn = new mysqli(localhost, $username, $password, $database);
    if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
    $query="SELECT * FROM stations ORDER BY StationName";
    $result=mysqli_query($conn, $query);
    $num=mysqli_num_rows($result);
    mysqli_close($conn);
    $i=1;
    $count=0;
    while ($i <= $num)
    {
        $row=mysqli_fetch_assoc($result);
        $s=$row["StationName"];
        $s=str_replace("*", " ", $s);
        $s=trim($s);
        if (!($stationList[$count]==$s)&&(strlen($s)>2))
        {
            $count++;
            $stationList[$count]=$s;
        }
        $i++;
    }
    return $stationList;
}
```

Save the **Stations.php** file and copy it to the server.

We can make a small improvement to the map display by linking multiple symbols representing the same underground station, as in the case of King's Cross, Euston and Baker Street in this example.



Open the **mapFunctions.php** file and locate the **displayStations( )** function. Add the block of program code below. Save the file and copy it to the server. Run the web page and check that lines are now drawn to connect the multiple symbols.

```

    if (show==true)
    {
        fill(r,g,b);
        stroke(0);
        xpos=int(xCentre)-int(transH);
        ypos=int(yCentre)-int(transV);
        ellipse(xpos,ypos,14,14);

        position=stationObj[i].position;
        if ((position==0)&&(midpoints=='NO'))
        {
            for (j=1;j<=stationCount ;j++ )
            {
                if (!(i==j))
                {
                    stationName2=stationObj[j].stationName;
                    if (stationName==stationName2)
                    {
                        xCentre=stationObj[j].xpos;
                        yCentre=stationObj[j].ypos;
                        xpos2=int(xCentre)-int(transH);
                        ypos2=int(yCentre)-int(transV);
                        fill(0);
                        line(xpos,ypos,xpos2,ypos2);
                    }
                }
            }
        }
    }
}

```

The next step is to create two drop-down lists showing the underground stations in alphabetical order. This will allow the user to select the start point and destination for their journey.

Return to the **index.php** file. Add two further lines of code to the PHP block at the beginning of the file.

```

include ('Stations.php');
$stationCount=Stations::loadStations();
include('RoutePoint.php');
$pointCount=RoutePoint::loadPoints();

$stationList=Stations::loadStationList();
$listCount =sizeof($stationList);

?>

```

Go to the end of the **<script>** block in **index.php** and add a **makeLists( )** function as shown below.

```

function makeLists(listCount,stationList)
{
    self = createSelect();
    self.position(1050, 80);
    self.size(250, 30);
    self.option(' ');
    for (i=1;i<=listCount;i++)
    {
        self.option(stationList[i]);
    }
    selT = createSelect();
    selT.position(1050, 150);
    selT.size(250, 30);
    selT.option(' ');
    for (i=1;i<=listCount;i++)
    {
        selT.option(stationList[i]);
    }
}
</script>

```

Move now to the **setup()** function near the start of the **<script>** block. Add lines of program code to produce captions for the drop-down list boxes, then call the function to create the lists.

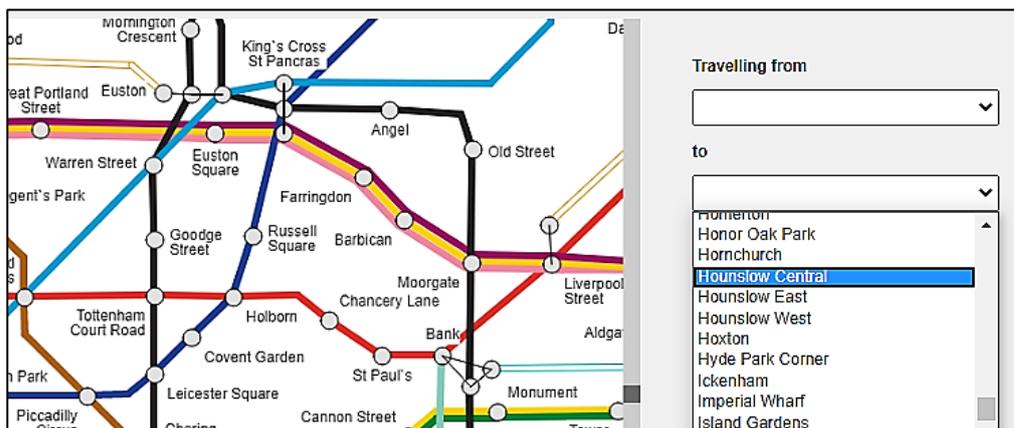
```

function setup()
{
    if ((Hscroll>0)|| (Vscroll>0))
    {
        VscrollPosition=Vscroll;
        HscrollPosition=Hscroll;
    }
    createCanvas(1000, 654);

    captionF = createElement('h3', 'Travelling from');
    captionF.position(1050, 40);
    captionT = createElement('h3', 'to');
    captionT.position(1050, 110);
    makeLists(listCount,stationList);
}

```

Save the **index.php** file and copy it to the server. Run the web page and check that stations are listed in the drop-down selection boxes, as in the example below.



Return to the **setup( )** function in the **index.php** file. Add lines of program code as shown below to create a 'clear' button.

```
captionF = createElement('h3', 'Travelling from');
captionF.position(1050, 40);
captionT = createElement('h3', 'to');
captionT.position(1050, 110);
makeLists(listCount,stationList);

buttonC = createButton('clear');
buttonC.position(1080, 200);
buttonC.mouseClicked(clearSelection);
}
```

Go now to the end of the **<script>** block and insert a **clearSelection( )** function. This will reset the two drop-down lists to show blank input boxes.

```
function clearSelection()
{
    selT.remove();
    selF.remove();
    makeLists(listCount,stationList);
}
```

Save the **index.php** file and copy it to the server. Run the web page and select stations from the drop-down lists, then click the 'clear' button. Check that the input boxes are cleared correctly.

As an alternative to selecting stations from the drop-down lists, the user will be able to select the start and finish points for their journey by clicking on station symbols on the map. The name of the first station clicked will be inserted into the 'Travelling from' drop-down list box. The name of the second station clicked will be inserted into the 'to' box.

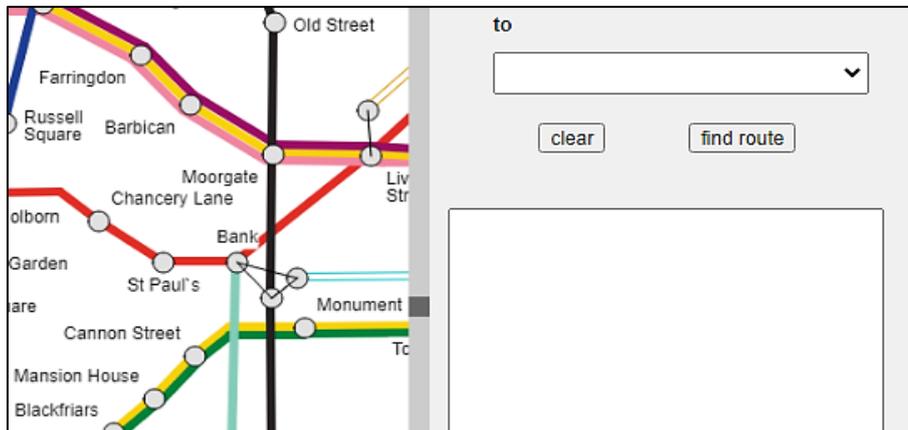
Return to the **index.php** file. Add the **mouseClicked( )** function below the **draw( )** function in the **<script>** block, as shown in the two boxes below.

```
function mouseClicked()
{
    x=mouseX;
    y=mouseY;
    if((x<960)&&(y<620))
    {
        if ((Vscroll==false)&&(Hscroll==false))
        {
            for (i=1;i<=stationCount ;i++ )
            {
                xCentre=int(stationObj[i].xpos);
                yCentre=int(stationObj[i].ypos);
                Xdiff = abs(xCentre-(x+transH));
                Ydiff = abs(yCentre-(y+transV));
                if ((Xdiff<10)&&(Ydiff<10))
                    stationID=stationObj[i].stationID;
            }
            selT.remove();
            selF.remove();
            makeLists(listCount,stationList);
        }
    }
}
```



Save the **index.php** file and copy it to the server. Run the web page. Select start and destination stations by clicking on the map, and check that these appear in the drop-down list boxes. Check also that the clear button cancels the entries correctly.

Now that the start and destination can be selected, either from the drop-down lists or by clicking on the map, we can move on to find routes between these stations. We will begin by adding a 'find route' **button** and a **text area** where the route information will be displayed.



Return to the **index.php** file and locate the **setup( )** function. Add lines of code as shown below to create the button and text area.

```

buttonC = createButton('clear');
buttonC.position(1080, 200);
buttonC.mouseClicked(clearSelection);

buttonR = createButton('find route');
buttonR.position(1180, 200);
buttonR.mouseClicked(findRoute);
textArea = createElement('textarea');
textArea.position(1020,260);
textArea.attribute("rows","24");
textArea.attribute("cols","40");
}

```

Go to the bottom of the **<script>** block and add a **findRoute( )** function below **clearSelection( )**. This function will be called when the 'find route' button is clicked. Save **index.php** and copy it to the server.

```

function findRoute()
{
    window.location = "findRoute.php?fromStation="+self.value()
        +"&toStation="+selfT.value()+"&Hscroll=" +int(HscrollPosition)+
        "&Vscroll="+int(VscrollPosition);
}

```

We will now create a new page which will be loaded when the 'find route' button is clicked. The names of the start and destination stations will be transferred to this page as part of the URL address. Open a blank file and add the program code below. Save the file as **findRoute.php** and copy it to the server.

```

<?
  $fromStation=$_REQUEST['fromStation'];
  $toStation=$_REQUEST['toStation'];
?>
<html>
<body>
<?
  $message=$fromStation." to ".$toStation;
  echo $message;
  echo"<form method=post action='index.php?message=".$message."'>";
  echo"<p><input type=submit value='continue'>";
  echo"</form>";
?>
</body>
</html>

```

The program creates a text string **\$message** which is made up from the names of the start and destination stations selected earlier. In the completed system, this message will also contain details of routes which have been found. It is displayed on the page for test purposes.

Leicester Square to Monument

The **\$message** string is transferred back to the **index.php** page when the 'continue' button is clicked. Return to **index.php** and add lines of code near the start to collect the message text.

```

include('RoutePoint.php');
$pointCount=RoutePoint::loadPoints();
$stationList=Stations::loadStationList();
$listCount =sizeof($stationList);

$message=$_REQUEST['message'];
if (!isset($message))
  $message=" ";
?>
<html>
<head>

```

At the start of the **<body>** section, add a block of css code to set the font for the text area.

```

<script src="p5.dom.js"></script>
</head>
<body>

  <style>
  textarea {
    font-family: Arial, Helvetica, sans-serif;
  }
  </style>

  <br>
  <script type="text/javascript">
    var VscrollPosition=300;

```

Near the start of the `<script>` block, add a line of program code to convert the PHP variable `$message` into a JavaScript variable as shown below.

```

var fromStation="";
var inputCount=0;
message=<? echo json_encode($message); ?>;
function setup()
{

```

Go now to the `setup( )` function and add lines of program code to display the message string in the text area. A slight problem is that the normal HTML `<br>` command for creating a new line of text does not work in a text area component. This must be replaced with the character sequence `'\n\r'`.

```

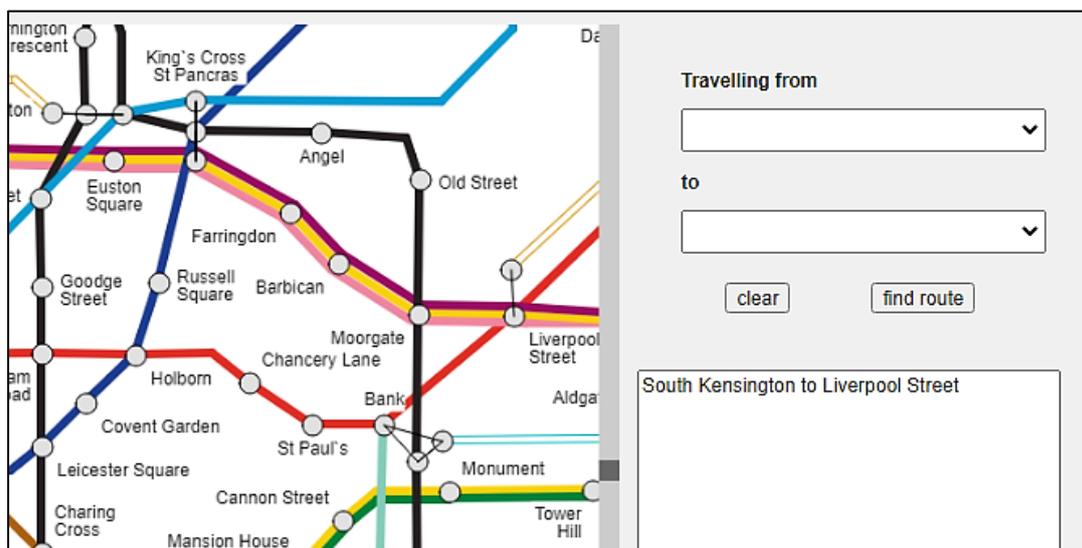
textArea = createElement('textarea');
textArea.position(1020,260);
textArea.attribute("rows","24");
textArea.attribute("cols","40");
message = message.replace(/<br>/g, '\n\r');
textArea.html(message);
}

```

Save the `index.php` file and copy it to the server.

Run the web page and select a start and destination station, either by means of the drop-down lists or by clicking the mouse on the map. Click the 'find route' button to go to the `findRoute.php` page. Check that the correct station names are displayed.

Click 'continue' to return to `index.php`. Check that the station names are now shown in the text area, as in the example below.



One final task is to add program code to clear the text area along with the drop-down list boxes when the 'clear' button is clicked. Return to the `index.php` file and locate the `clearSelection( )` function. Add lines of program code as shown below.

```
function clearSelection()
{
    self.remove();
    self.remove();
    makeLists(listCount,stationList);
    inputCount=0;
    fromStation="";

textarea.remove();
        textarea = createElement('textarea');
        textarea.position(1020,260);
        textarea.attribute("rows","24");
        textarea.attribute("cols","40");


}

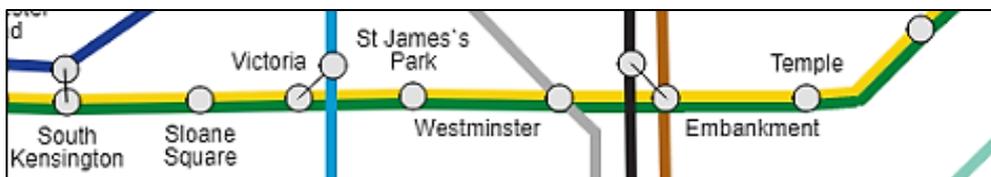
```

Save the **index.php** file and copy it to the server. Run the web page, enter stations, then click the 'find route' button. Click 'continue' to return to the index.php page and display the message string in the text area. Check that the message text is deleted when the 'clear' button is clicked.

This completes the design of the user interface. We will now work on the **findRoute.php** page which will run the algorithm for finding routes between the start and destination stations.

The strategy for finding routes can be summarised as follows:

- We first search for a connection which does not involve a change of underground line. To do this, it is necessary to produce a list of stations for each line in turn, and check whether both the departure and destination points are present in the same list. It may be the case that more than one line provides a direct connection between stations, as with the **District** and **Circle** lines between **South Kensington** and **Temple**:



The lists should be made up from station names rather than stationID numbers. This will ensure that multiple station points with different ID numbers, as at **Victoria** and **Embankment** in the above example, are recognised as the same station.

- If no direct link is found, then we search for journeys involving one change of underground line. Each line serving the start station is compared with each line serving the destination. If a station is found which is on both a start and finish line, then this is a possible change station.
- If a connection is not found with one change of underground line, we search for a connection involving two changes. A list is made of all stations which can be reached from the starting point with one change. A check is made for an underground line running from any of these stations to the destination.
- Due to the highly interconnected nature of the London Underground system, it was found that a journey could be made between any two stations on the network with a maximum of two changes.

All the necessary data for finding routes can be obtained from the **line**, **station** and **routePoint** tables, as in the example below.

To find the stations on a particular underground line, such as the **Central** line shown here, we use the **lineID** value to identify all route points on that line. The names of the stations can then be found from the **stationID** value. At this stage we are not concerned about the order in which the stations occur along the line, but only that they lie on the same underground line.

routePointID	lineID	stationID	pointer	backpointer	branch	position
7	1	235	8	6	1	0
8	1	121	20	7	1	0
9	1	2	24	10	1	0
10	1	97	9	11	1	0
11	1	9				
12	1	9				

StationID	StationName	Xpos	Ypos	Position
1	Tottenham*Court Road	1237	806	7
2	Oxford*Circus	1131	807	1
3	Camden Town*	1279	544	8
4	Swiss Cottage*	1014	588	3
		1495	1012	5

lineID	lineName	colourCode	solid
1	Central	227,44,36	1
2	Northern	36,32,33	1
3	Circle	248,210,8	1

Notice that some **stationID** values show no station name. These represent points which were inserted between stations simply to allow the line to follow a more accurate route on the map.

We will begin by creating a function to obtain all station names for points along a specified underground line. Go to the **RoutePoint.php** class file and add a **stationList( )** method as shown below. This uses an SQL command to obtain the **stationIDs**, then another SQL command to obtain the corresponding station names. The results are returned as a **\$stationName** array.

```

public static function stationList($lineID)
{
    include ('user.inc');
    $conn = new mysqli(localhost, $username, $password, $database);
    if (!$conn) {die("Connection failed: ".mysqli_connect_error()); }
    $query="SELECT stationID FROM routePoint WHERE lineID='$lineID'";
    $result=mysqli_query($conn, $query);
    $num=mysqli_num_rows($result);
    $i=1;
    $count=0;
    while ($i <= $num)
    {
        $row=mysqli_fetch_assoc($result);
        $station=$row["stationID"];
        $stationQuery="SELECT StationName FROM stations WHERE
                    StationID='$station'";
        $stationResult=mysqli_query($conn, $stationQuery);
        $row=mysqli_fetch_assoc($stationResult);
        $name=$row["StationName"];
        if (strlen($name)>2)
        {
            $stationName[$count]=trim(str_replace("*", " ", $name));
            $count++;
        }
        $i++;
    }
    mysqli_close($conn);
    return $stationName;
}
}
?>

```

Save the **RoutePoint.php** file and copy it to the server.

Return to **findRoute.php** and add lines of program code as shown below. The 'include' commands will allow access to the RoutePoint, Line and Stations classes. A loop is then used to obtain station lists for each of the underground lines in turn. Checks are carried out to determine whether both '**fromStation**' and '**toStation**' are present in the same list, indicating that the journey can be carried out without changing line.

```

<?
  $fromStation=$_REQUEST['fromStation'];
  $toStation=$_REQUEST['toStation'];

  include('RoutePoint.php');
  include('Line.php');
  include ('Stations.php');
  $routeResultCount=0;
  $routeResult=array();
  $routeResult[] = array();

?>
<html>
<body>
<?
  $message=$fromStation." to ".$toStation;
  echo $message;

  $lineCount=Line::loadLines();
  for ($j=1;$j<=$lineCount;$j++)
  {
    $lineID=Line::$lineObj[$j]->lineID;
    $fromFound=false;
    $toFound=false;
    $list=RoutePoint::stationList($lineID);
    for ($i=0;$i<=count($list);$i++)
    {
      if ($list[$i]==$fromStation)
      {
        $fromFound=true;
      }
      if ($list[$i]==$toStation)
      {
        $toFound=true;
      }
    }
    if (($fromFound==true)&&($toFound==true))
    {
      $lineName=Line::getLineName($lineID);
      $routeResultCount++;
      $routeResult[$routeResultCount][1]=$fromStation;
      $routeResult[$routeResultCount][2]=$toStation;
      $routeResult[$routeResultCount][3]=$lineName;
    }
  }

  echo"<form method=post action='index.php?message=".$message."'>";
  echo"<p><input type=submit value='continue'>";
  echo"</form>";

?>
</body>

```

Results will be displayed in a table. Continuing to work in the **findRoute.php** file, insert lines of code to produce table headings, then a loop to output the details of routes which have been found.

Save **findRoute.php** and copy it to the server.

```

        $routeResult[$routeResultCount][2]=$toStation;
        $routeResult[$routeResultCount][3]=$lineName;
    }
}

echo"<p>";
echo"<table border=1 cellpadding=5>";
echo"<tr><th></th>";
echo"<th>From station</th>";
echo"<th>To station</th>";
echo"<th>First line</th>";
for ($i=1;$i<=$routeResultCount;$i++)
{
    echo"<tr><td>".$i."</td>";
    echo"<td>".$routeResult[$i][1]."</td>";
    echo"<td>".$routeResult[$i][2]."</td>";
    echo"<td>".$routeResult[$i][3]."</td>";
}
echo"</table>";

echo"<form method=post action='index.php?message=".$message.">";
echo"<p><input type=submit value='continue'>";

```

Open the **Line.php** class file and add a small method as shown below. This will take the ID number for an underground line as the input parameter, then return the corresponding line name. Save **Line.php** and copy it to the server.

```

public static function getLineName($lineID)
{
    $lineCount = Line::loadLines();
    for($i=1;$i<=$lineCount;$i++)
    {
        if (Line::$lineObj[$i]->lineID == $lineID)
            $lineName = Line::$lineObj[$i]->lineName;
    }
    return $lineName;
}
}
?>

```

Run the web site. Carry out tests by entering a series of journeys which can be made without changing underground line. In each case, check that the program selects the correct line(s) as in the examples shown

South Kensington to Monument

	From station	To station	First line
1	South Kensington	Monument	Circle
2	South Kensington	Monument	District

South Kensington to King's Cross St Pancras

	From station	To station	First line
1	South Kensington	King's Cross St Pancras	Circle
2	South Kensington	King's Cross St Pancras	Piccadilly

We can now move on to consider journeys where one change of line is necessary. The strategy will be to keep a list of the underground lines serving the start station, and another list of the lines serving the destination. Stations along these lines can then be compared to find a change point.

Return to the **findRoute.php** file. Go to the loop structure and add lines to the two 'if...' blocks. These will collect the **lineID** values for any lines passing through the start or destination stations and store them in arrays.

```

    $toFound=false;
    $list=RoutePoint::stationList($lineID);
    for ($i=0;$i<=count($list);$i++)
    {
        if ($list[$i]==$fromStation)
        {
            $fromFound=true;

            $fromCount++;
            $fromArray[$fromCount]=$lineID;
        }
        if ($list[$i]==$toStation)
        {
            $toFound=true;

            $toCount++;
            $toArray[$toCount]=$lineID;
        }
    }
    if (($fromFound==true)&&($toFound==true))

```

Add three variables near the start of the **<body>** section.

```

$message=$fromStation." to ".$toStation;
echo $message;

$direct=false;
$fromCount=0;
$toCount=0;

$lineCount=Line::loadLines();

```

The Boolean variable **\$direct** will be set to true or false, depending on whether a direct connection is found from the start point to the destination with no change of line. The integer variables **\$fromCount** and **\$toCount** will count the number of different underground lines serving the start station and the destination.

Move down towards the end of the **<body>** section and add the line of program code shown below. The value of **\$direct** is set to **true** if a direct connection is found.

```

    if (($fromFound==true)&&($toFound==true))
    {
        $direct=true;

        $lineName=Line::getLineName($lineID);
        $routeResultCount++;
        $routeResult[$routeResultCount][1]=$fromStation;
    }

```

Move down and add the block of program code shown below. This code will operate only if a change underground line is necessary.

```

        $routeResult[$routeResultCount][2]=$toStation;
        $routeResult[$routeResultCount][3]=$lineName;
    }
}

if ($direct==false)
{
    $resultCount= oneChange($fromStation,$toStation,$fromArray,
                            $toArray,$fromCount,$toCount);
    if ($resultCount==0)
    {
        echo"<p>No connection found";
    }
}

echo"<p>";
echo"<table border=1 cellpadding=5>";

```

The program will call a function **oneChange()**. Set up this function at the end of the **findRoute.php** file, below the closing `</body>` and `</html>` tags.

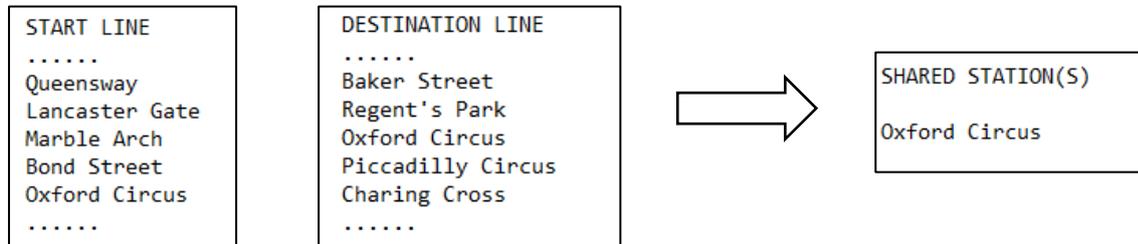
```

</body>
</html>

<?
function oneChange($fromStation,$toStation,$fromArray, $toArray,
                  $fromCount,$toCount)
{
    $lineCount=Line::loadLines();
    $stationCount=Stations::loadStations();
    global $resultArray;
    global $routeResult;
    global $routeResultCount;
    $resultCount=0;
    for ($i=1;$i<=$fromCount;$i++)
    {
        $lineName1=Line::getLineName($fromArray[$i]);
        $list1=RoutePoint::stationList($fromArray[$i]);
        for ($j=1;$j<=$toCount;$j++)
        {
            $lineName2=Line::getLineName($toArray[$j]);
            $list2=RoutePoint::stationList($toArray[$j]);
            $count=checkConnection($list1,$list2,$lineName1,$lineName2);
            for ($k=1;$k<=$count;$k++)
            {
                $routeResultCount++;
                $routeResult[$routeResultCount][1]=$fromStation;
                $routeResult[$routeResultCount][2]=$toStation;
                $routeResult[$routeResultCount][3]=$lineName1;
                $routeResult[$routeResultCount][4]=$resultArray[$k];
                $routeResult[$routeResultCount][5]=$lineName2;
            }
            $resultCount= $resultCount+$count;
        }
    }
    return $resultCount;
}
?>

```

The function takes each of the lines serving the start station in turn, and compares it to each of the lines serving the destination. This is done by calling a function **checkConnection( )**. The function returns a list of any stations which are on both of the lines. A journey could then be made by travelling on the first line, changing at any shared station, then completing the journey on the second line, e.g.:



Add the **checkConnection( )** and **getStationName( )** functions at the end of the **findRoute.php** page as shown below. The **checkConnection( )** function makes use of an **array\_intersect()** command. This very useful PHP function takes two arrays as its input parameters, then creates a results array containing only the items which are common to both of the input arrays.

```
function checkConnection($list1,$list2,$lineName1,$lineName2)
{
    global $resultArray;
    $intersection = array_intersect($list1, $list2);
    $resultCount=0;
    for ($k=0;$k<count($list1);$k++)
    {
        if (strlen($intersection[$k])>1)
        {
            $resultCount++;
            $resultArray[$resultCount]=$intersection[$k];
        }
    }
    return $resultCount;
}

function getStationName($stationID, $stationCount)
{
    for($i=1;$i<=$stationCount;$i++)
    {
        if (Stations::$stationObj[$i]->stationID == $stationID)
        {
            $stationName = Stations::$stationObj[$i]->stationName;
            $stationName=trim(str_replace("*", " ", $stationName));
        }
    }
    return $stationName;
}

```

?>

Add lines of code to add extra columns to extend the results array, as shown below.

```

echo"<th>From station</th>";
echo"<th>To station</th>";
echo"<th>First line</th>";

echo"<th>First change</th>";
echo"<th>Second line</th>";

for ($i=1;$i<=$routeResultCount;$i++)
{
    echo"<tr><td>.$i.</td>";
    echo"<td>.$routeResult[$i][1].</td>";
    echo"<td>.$routeResult[$i][2].</td>";
    echo"<td>.$routeResult[$i][3].</td>";

    echo"<td>.$routeResult[$i][4].</td>";
    echo"<td>.$routeResult[$i][5].</td>";

}
echo"</table>";

```

Save **findRoute.php** and copy it to the server. Run the website and select journeys which can be made with one change of underground line. Check that all route options are displayed correctly. Also check that journeys requiring more than one change of underground line display the message 'No connection found'.

Queensway to Piccadilly Circus

	From station	To station	First line	First change	Second line
1	Queensway	Piccadilly Circus	Central	Oxford Circus	Bakerloo
2	Queensway	Piccadilly Circus	Central	Holborn	Piccadilly

Mitcham Junction to Brixton

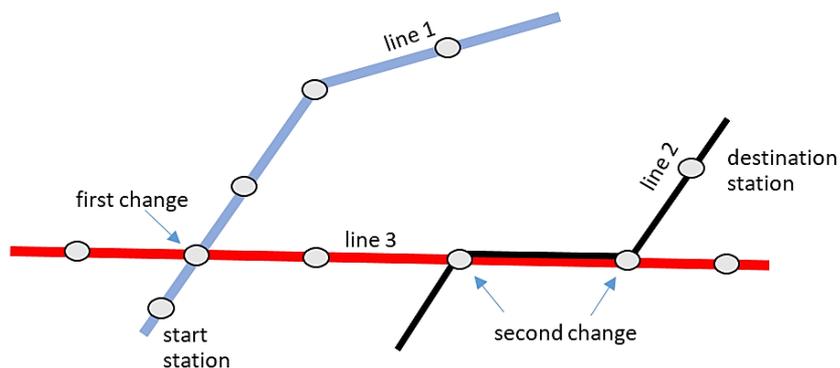
No connection found

When testing the program, you may discover that some routes are duplicated by the search algorithm. We will return to correct this problem later.

We will now consider journeys involving two changes of underground line. Create a **twoChanges()** function in the **findRoute.php** file by adding the block of code shown on the next page.

The start station may be served by just one line, as in the case of Russel Square, or up to a total of six different lines in the case of King's Cross. Any line passing through the start station is selected, for example the Victoria line, and this is designated as **line1**. A list of all stations along this line is compiled.

The destination station may again be served by one or more lines. Any line passing through the destination station is selected, for example the Northern line, and this is designated as **line2**. A list of all stations along this line is again compiled.



```

function twoChanges($fromStation,$toStation,$fromArray, $toArray,$fromCount,$toCount)
{
    $lineCount=Line::loadLines();
    $stationCount=Stations::loadStations();
    global $resultArray;
    global $routeResult;
    global $routeResultCount;
    $resultCount=0;
    for ($i=1;$i<=$fromCount;$i++)
    {
        $lineName1=Line::getLineName($fromArray[$i]);
        $list1=RoutePoint::stationList($fromArray[$i]);
        for ($j=1;$j<=$toCount;$j++)
        {
            $lineName2=Line::getLineName($toArray[$j]);
            $list2=RoutePoint::stationList($toArray[$j]);
            for ($h=1;$h<=$lineCount;$h++)
            {
                $lineID = Line::$lineObj[$h]->lineID;
                if (($lineID != $fromArray[$i])&&($lineID != $toArray[$j]))
                {
                    $list3=RoutePoint::stationList($lineID);
                    $lineName3=Line::getLineName($lineID);
                    $result1=checkConnection($list1,$list3,$lineName1,$lineName3);
                    $resultArray1=$resultArray;
                    $result2=checkConnection($list2,$list3,$lineName2,$lineName3);
                    $resultArray2=$resultArray;
                    if (($result1>=1)&&($result2>=1))
                    {
                        for ($k=1;$k<=$result1;$k++)
                        {
                            for ($w=1;$w<=$result2;$w++)
                            {
                                $routeResultCount++;
                                $routeResult[$routeResultCount][1]=$fromStation;
                                $routeResult[$routeResultCount][2]=$toStation;
                                $routeResult[$routeResultCount][3]=$lineName1;
                                $routeResult[$routeResultCount][4]=$resultArray1[$k];
                                $routeResult[$routeResultCount][5]=$lineName3;
                                $routeResult[$routeResultCount][6]=$resultArray2[$w];
                                $routeResult[$routeResultCount][7]=$lineName2;
                                $resultCount++;
                            }
                        }
                    }
                }
            }
        }
    }
    return $resultCount;
}

```

We now search for a third line which connects lines 1 and 2 and would form the central link of a journey involving two changes. We choose a line different to lines 1 and 2, such as the Central line. This is designated as **line3** and a list of all stations along the line is again compiled.

The **checkConnection()** function is called to determine whether one or more stations are on both **line 1** and **line 3**. It is also called to determine whether any stations are on both **line 2** and **line 3**. If shared stations are found in both cases, then a route has been found. The names of lines 1,2 and 3 are recorded, along with the names of the possible change stations.

The program then uses loops to check other start, finish and connecting lines, so that all routes from the start station to the destination station involving two changes of underground line are found.

We again need to extend the results array to display the additional data. Add code as shown below.

```

echo"<th>First change</th>";
echo"<th>Second line</th>";

echo"<th>Second change</th>";
echo"<th>Third line</th>";

for ($i=1;$i<=$routeResultCount;$i++)
{
    echo"<tr><td>". $i. "</td>";
    echo"<td>". $routeResult[$i][1]. "</td>";
    echo"<td>". $routeResult[$i][2]. "</td>";
    echo"<td>". $routeResult[$i][3]. "</td>";
    echo"<td>". $routeResult[$i][4]. "</td>";
    echo"<td>". $routeResult[$i][5]. "</td>";

    echo"<td>". $routeResult[$i][6]. "</td>";
    echo"<td>". $routeResult[$i][7]. "</td>";

}
echo"</table>";

```

Continuing to work in the **findRoute.php** file, insert program lines to call the **twoChanges( )** function.

```

if ($direct==false)
{
    $resultCount=oneChange($fromStation,$toStation,$fromArray,
                           $toArray,$fromCount,$toCount);

    if ($resultCount==0)
    {
        $resultCount=twoChanges($fromStation,$toStation,$fromArray,
                                $toArray,$fromCount,$toCount);
    }

    if ($resultCount==0)
    {
        echo"<p>No connection found";
    }
}

```

Save the **findRoute.php** file and copy it to the server.

Run the website and select journeys which require more than one change of underground line. Check that all route options are displayed correctly.

Brixton to Lewisham

	From station	To station	First line	First change	Second line	Second change	Third line
1	Brixton	Lewisham	Victoria	Oxford Circus	Central	Bank	DLR
2	Brixton	Lewisham	Victoria	Oxford Circus	Central	Stratford	DLR
3	Brixton	Lewisham	Victoria	Oxford Circus	Central	Stratford	DLR
4	Brixton	Lewisham	Victoria	Victoria	District	West Ham	DLR
5	Brixton	Lewisham	Victoria	King's Cross St Pancras	Hammersmith and City	West Ham	DLR
6	Brixton	Lewisham	Victoria	Green Park	Jubilee	Canning Town	DLR
7	Brixton	Lewisham	Victoria	Green Park	Jubilee	Canary Wharf	DLR

When testing the program, you may discover that some routes are duplicated by the search algorithm, as in the case of options 2 and 3 listed above. This problem can be corrected by including a **removeDuplicates()** function.

Return to the **findRoute.php** file. Add a line of code to call **removeDuplicates()** before the results table is displayed.

```

    if ($resultCount==0)
    {
        echo"<p>No connection found";
    }
}

$routeResult=removeDuplicates($routeResult,$routeResultCount);

echo"<p>";
echo"<table border=1 cellpadding=5>";
echo"<tr><th></th>";
echo"<th>From station</th>";

```

Go to the end of the **findRoute.php** file and add the **removeDuplicates()** function, as shown below. Save **findRoute.php** and copy it to the server.

```

function removeDuplicates($routeResult,$routeResultCount)
{
    global $routeResultCount;
    for ($i=1;$i<=$routeResultCount;$i++)
    {
        $c[$i]=$routeResult[$i][1]." ".$routeResult[$i][2]." "
            . $routeResult[$i][3]." ";
        $c[$i]=$c[$i].$routeResult[$i][4]." ".$routeResult[$i][5]." ";
        $c[$i]=$c[$i].$routeResult[$i][6]." ".$routeResult[$i][7];
        $routeResult[$i][10]=0;
    }
    for ($i=2;$i<=$routeResultCount;$i++)
    {
        $current=$c[$i];
        for ($j=1;$j<$i;$j++)
        {
            if ($c[$j]==$current)
            {
                $routeResult[$i][10]=-1;
                $routeResult[$i][8]=-1;
            }
        }
    }
    $tempCount=0;
    for ($i=1;$i<=$routeResultCount;$i++)
    {
        if ($routeResult[$i][10]==0)
        {
            $tempCount++;
            $tempArray[$tempCount]=$routeResult[$i];
        }
    }
    $routeResult=$tempArray;
    $routeResultCount=$tempCount;
    return $routeResult;
}

```

The function operates by making up a text string for each row of the results table by combining the station and underground line entries in each of the columns. This string is then compared with the text strings produced for all previous rows, and the current row is marked for deletion if the exact same set of route data is found.

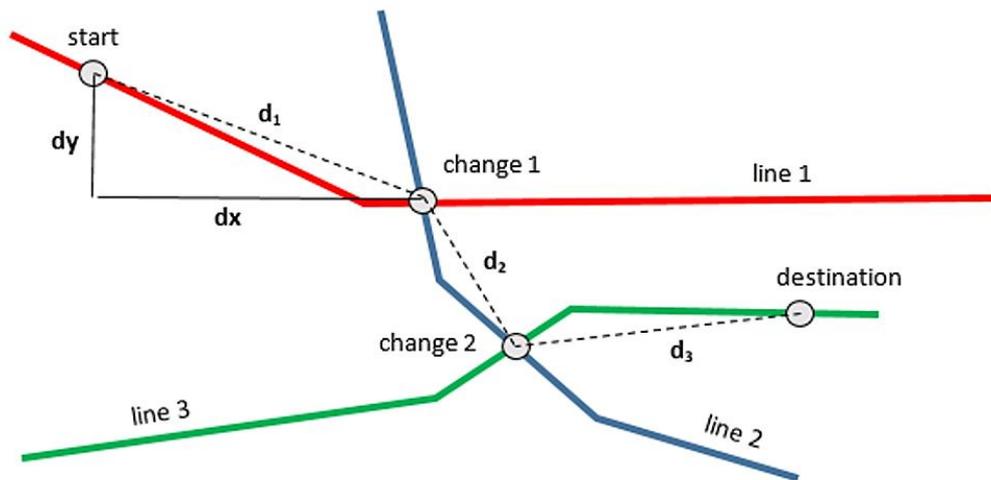
Run the website. Select journeys involving one or two changes of underground line, and check that there are now no duplicated routes in the results table.

Brixton to Lewisham							
	From station	To station	First line	First change	Second line	Second change	Third line
1	Brixton	Lewisham	Victoria	Oxford Circus	Central	Bank	DLR
2	Brixton	Lewisham	Victoria	Oxford Circus	Central	Stratford	DLR
3	Brixton	Lewisham	Victoria	Victoria	District	West Ham	DLR
4	Brixton	Lewisham	Victoria	King's Cross St Pancras	Hammersmith and City	West Ham	DLR

When testing the program with journeys involving changes of underground line, you will discover that a large number of different routes may be displayed. Some routes, whilst theoretically possible, may be much longer than others. We will recommend that the traveller chooses the route with the least number of intermediate stations on the assumption that this will be the shortest and most direct route.

Counting the intermediate stations between two points may not be a straightforward task. As we saw in the introduction to this chapter, the London Underground system has some lines which include loops, and some lines with multiple branches. As a consequence, the processing time required may be significant for a long list of alternative routes. To reduce the processing time, the more unsuitable routes can be quickly filtered out by estimating route lengths on the map display.

Consider a journey involving two changes of underground line. Straight line distances on the map between journey points can be found by means of Pythagoras' theorem:



The straight line distance from the start station to the first change station is given by:

$$d_1 = \sqrt{dx^2 + dy^2}$$

where **dx** is the difference in x coordinates between the station points, and **dy** is the difference in the y coordinates. After calculating the straight line lengths **d<sub>1</sub>**, **d<sub>2</sub>** and **d<sub>3</sub>** for the stages of the journey, the results can be added to obtain an estimate for the whole journey distance measured in screen pixels.

Open the **Stations.php** class file and add the **getPixels( )** method shown below. Save **Stations.php** and copy it to the server.

The **getPixels( )** method takes two station names as input parameters. A loop then checks each Station object to obtain the x- and y-coordinates of the stations. Pythagoras' formula is then used to calculate the straight line distance between the stations.

```
public static function getPixels($fromStation,$toStation)
{
    $stationCount=Stations::loadStations();
    $distance = 0;
    for ($i=1;$i<=$stationCount;$i++)
    {
        $s=Stations::$stationObj[$i]->stationName;
        $s=str_replace("*"," ",$s);
        $s=trim($s);
        if ($s==$fromStation)
        {
            $fromX=Stations::$stationObj[$i]->xpos;
            $fromY=Stations::$stationObj[$i]->ypos;
        }
        if ($s==$toStation)
        {
            $toX=Stations::$stationObj[$i]->xpos;
            $toY=Stations::$stationObj[$i]->ypos;
        }
    }
    $distance = sqrt(((($fromX-$toX)*($fromX-$toX))+((($fromY-$toY)*($fromY-$toY))));
    return $distance;
}
```

Return to the **findRoute.php** file. Locate the **oneChange( )** function and add lines of program code as shown below.

```
$count=checkConnection($list1,$list2,$lineName1, $lineName2);
for ($k=1;$k<=$count;$k++)
{
    $routeResultCount++;
    $routeResult[$routeResultCount][1]=$fromStation;
    $routeResult[$routeResultCount][2]=$toStation;
    $routeResult[$routeResultCount][3]=$lineName1;
    $routeResult[$routeResultCount][4]=$resultArray[$k];
    $routeResult[$routeResultCount][5]=$lineName2;

    $firstStage=Stations::getPixels($fromStation,$resultArray[$k]);
    $secondStage=Stations::getPixels($toStation,$resultArray[$k]);
    $distance = intval($firstStage+$secondStage);
    $routeResult[$routeResultCount][8]=$distance;
}
$resultCount= $resultCount+$count;
}
return $resultCount;
}
```

Continuing to work in the **findRoute.php** file, move up to the section where the results table is displayed. Add a further column heading and data output, as shown below.

```

echo"<th>Second change</th>";
echo"<th>Third line</th>";

echo"<th>Map pixels</th>";

for ($i=1;$i<=$routeResultCount;$i++)
{
    if (($resultCount<=4)||($routeResult[$i][8]>0))
    {
        echo"<tr><td>".$i."</td>";
        echo"<td>".$routeResult[$i][1]."</td>";
        echo"<td>".$routeResult[$i][2]."</td>";
        ....
        echo"<td>".$routeResult[$i][6]."</td>";
        echo"<td>".$routeResult[$i][7]."</td>";

        echo"<td>".$routeResult[$i][8]."</td>";

    }
}
echo"</table>";

```

Save the **findRoute.php** file and copy it to the server.

Run the web site and select several journeys involving one change of underground line. Distances should be shown in a 'Map pixels' column in the results table. Compare the lengths of the routes on the underground map and confirm that the estimates seem reasonable.

South Kensington to Bond Street

	From station	To station	First line	First change	Second line	Second change	Third line	Map pixels
1	South Kensington	Bond Street	Circle	Notting Hill Gate	Central			358
2	South Kensington	Bond Street	Circle	Liverpool Street	Central			1181
3	South Kensington	Bond Street	Circle	Westminster	Jubilee			517
4	South Kensington	Bond Street	Circle	Baker Street	Jubilee			433
5	South Kensington	Bond Street	District	Ealing Broadway	Central			1192
6	South Kensington	Bond Street	District	Notting Hill Gate	Central			358

Return to the **findRoute.php** file and locate the **twoChanges( )** function. Add program code as shown below. This calls the **getPixels( )** method three times, for each stage of the journey, then adds the results.

```

$routeResult[$routeResultCount][5]=$lineName3;
$routeResult[$routeResultCount][6]=$resultArray2[$w];
$routeResult[$routeResultCount][7]=$lineName2;

    $firstStage=Stations::getPixels($fromStation,$resultArray1[$k]);
    $secondStage=Stations::getPixels($resultArray1[$k],$resultArray2[$w]);
    $thirdStage=Stations::getPixels($toStation,$resultArray2[$w]);
    $distance = intval($firstStage+$secondStage+$thirdStage);
    $routeResult[$routeResultCount][8]=$distance;

    $resultCount++;
}
}

```

Save **findRoute.php** and copy it to the server. Run the web site. Select several journeys requiring two changes of underground line. Again check that the estimated distances in the 'Map pixels' column seem reasonable for the routes shown.

London City Airport to Blackhorse Road								
	From station	To station	First line	First change	Second line	Second change	Third line	Map pixels
1	London City Airport	Blackhorse Road	DLR	Bank	Central	Oxford Circus	Victoria	1971
2	London City Airport	Blackhorse Road	DLR	Stratford	Central	Oxford Circus	Victoria	2295
3	London City Airport	Blackhorse Road	DLR	West Ham	District	Victoria	Victoria	2402
4	London City Airport	Blackhorse Road	DLR	West Ham	Hammersmith and City	King's Cross St Pancras	Victoria	1691
5	London City Airport	Blackhorse Road	DLR	Canning Town	Jubilee	Green Park	Victoria	2135
6	London City Airport	Blackhorse Road	DLR	Canary Wharf	Jubilee	Green Park	Victoria	2093
7	London City Airport	Blackhorse Road	DLR	Stratford	Jubilee	Green Park	Victoria	2454

Go to the end of the **findRoute.php** file and add the **compareDistances( )** function shown on the next page.

A loop checks through the list of routes, picking out the row with the largest map pixel value. The pixel entry on this row is then replaced by a rogue value of -1.

	From station	To station	First line	First change	Second line	Second change	Third line	Map pixels
1	London City Airport	Blackhorse Road	DLR	Bank	Central	Oxford Circus	Victoria	1971
2	London City Airport	Blackhorse Road	DLR	Stratford	Central	Oxford Circus	Victoria	2295
3	London City Airport	Blackhorse Road	DLR	West Ham	District	Victoria	Victoria	-1
4	London City Airport	Blackhorse Road	DLR	West Ham	Hammersmith and City	King's Cross St Pancras	Victoria	1691
5	London City Airport	Blackhorse Road	DLR	Canning Town	Jubilee	Green Park	Victoria	2135

The procedure is repeated, setting the highest remaining pixel value to -1 on each pass, until only the four routes with the shortest estimated distances remain unaltered.

```
function compareDistances($routeResult,$routeResultCount)
{
    for ($j=1;$j<=($routeResultCount-4);$j++)
    {
        $max=0;
        for ($i=1;$i<=$routeResultCount;$i++)
        {
            if ($routeResult[$i][8]>0)
            {
                if ($routeResult[$i][8]>$max)
                {
                    $max=$routeResult[$i][8];
                    $maxpos=$i;
                }
            }
        }
        $routeResult[$maxpos][8]= -1;
    }
    return $routeResult;
}
```

?>

Move to the section of the **findRoute.php** file where the results table is displayed. Call the **compareDistances ( )** function and add an **if..** conditional block so that only the routes left unaltered by the **compareDistances( )** function are displayed.

```

echo"<th>Map pixels</th>";
$routeResult=compareDistances($routeResult,$routeResultCount);
for ($i=1;$i<=$routeResultCount;$i++)
{
    if (($resultCount<=4)||($routeResult[$i][8]>0))
    {
        echo"<tr><td>".$i."</td>";
        echo"<td>".$routeResult[$i][1]."</td>";
        echo"<td>".$routeResult[$i][2]."</td>";
        .....
        echo"<td>".$routeResult[$i][7]."</td>";
        echo"<td>".$routeResult[$i][8]."</td>";
    }
}
echo"</table>";

```

Save the **findRoute.php** file and copy it to the server. Run the web site and select a variety of journeys involving one or two changes of underground line. In each case, no more than four possible route options should be listed.

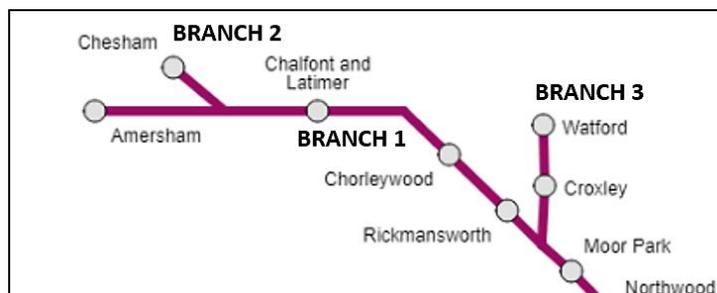
Hounslow Central to London City Airport

	From station	To station	First line	First change	Second line	Second change	Third line	Map pixels
1	Hounslow Central	London City Airport	Piccadilly	Holborn	Central	Bank	DLR	2066
16	Hounslow Central	London City Airport	Piccadilly	Green Park	Jubilee	Canary Wharf	DLR	2032
18	Hounslow Central	London City Airport	Piccadilly	Green Park	Jubilee	Canning Town	DLR	2074
21	Hounslow Central	London City Airport	Piccadilly	Leicester Square	Northern	Bank	DLR	2043

The next step is to count the number of intermediate stations along each route. A preferred route can then be selected with the least intermediate stations, and an alternative route selected with the second-least number of intermediate stations.

A new class will be created with the name '**Branch**'. This differs from the other classes in the program as it is not linked to a particular database table. Instead, **Branch** will obtain its data from the **routePoint**, **station** and **line** tables as necessary.

A **Branch object** will represent each separate branch of an underground line. The line may be simple with all stations on a single branch, or may have a number of branches serving different destinations.



Open a new file and add the program code shown below. Save the file as **Branch.php**.

```
<?
class Branch
{
    public static $branchObj = array();
    private $branchID;
    private $lineID;
    private $stationID = array();
    private $stationName = array();
    private $pointCount;

    function __construct($branchID,$lineID,$pointCount,$stationID, $stationName)
    {
        $this->branchID = $branchID;
        $this->lineID = $lineID;
        $this->pointCount = $pointCount;
        $this->stationID = $stationID;
        $this->stationName = $stationName;
    }
}
?>
```

The **Branch objects** will be constructed by accessing linked lists of stations stored in the **RoutePoint** database table. Each linked list, as in the example below, represents the sequence of station points along a branch of an underground line.

routePointID	lineID	stationID	pointer	backpointer	
1	2		↓	-1	A
3	2		↓		
4	2		↓		
5	2		↓		
6	2		-1		
7	4		-1		B1
8	4		↑		
9	4		↑		
10	4			-1	
12	5		↓	-1	C
13	5		↓		
15	5		↓		
17	4		-1	-1	B2
18	4				
22	5		↓		C
24	5		↓		
25	5		↓	-1	

- Each linked list begins with a **backpointer** set to -1, and ends with a **pointer** set to -1. The linked list is accessed in sequence by means of pointers which refer to **routePointID** values.
- A linked list may move forwards (e.g. list A) or backwards (e.g. list B1) through the set of records in the database table.
- An underground line with several branches will appear as separate linked lists (e.g. B1 and B2).
- The linked list may be split across several blocks of records, depending on the pattern in which the data was entered, (e.g. list C). The sequence can, however, be easily followed from start to finish by means of the pointer values.

Continuing to work on the **Branch.php** file, add the **loadBranch( )** method shown below.

```

public static function loadBranch($lineIDwanted)
{
    echo"<p>Loading branches for line ".$lineIDwanted;
    $pointCount = RoutePoint::loadPoints();
    $stationCount = Stations::loadStations();
    $count=0; $objectCount=0;
    $stationID = array();
    $stationName = array();
    for ($n=1;$n<=$pointCount;$n++)
    {
        if (RoutePoint::$pointObj[$n]->backpointer==-1)
        {
            $lineID=RoutePoint::$pointObj[$n]->lineID;
            if ($lineID==$lineIDwanted)
            {
                $count++;
                $routePointID=RoutePoint::$pointObj[$n]->routePointID;
                $stationID[$count]=RoutePoint::$pointObj[$n]->stationID;
                $stationName[$count]=Branch::getStationName($stationID[$count],
                    $stationCount);
                $pointer=RoutePoint::$pointObj[$n]->pointer;
                $next=$pointer;
                $finished=false;
                while ($finished==false)
                {
                    for ($m=1;$m<=$pointCount;$m++)
                    {
                        $routePointID=RoutePoint::$pointObj[$m]->routePointID;
                        if ($routePointID==$next)
                        {
                            $count++;
                            $stationID[$count]=RoutePoint::$pointObj[$m]->stationID;
                            $stationName[$count]=
                                Branch::getStationName($stationID[$count],$stationCount);
                            $next=RoutePoint::$pointObj[$m]->pointer;
                        }
                    }
                    if ($next== -1)
                    {
                        $objectCount++;
                        echo"<p>Branch ".$objectCount;
                        for ($s=1;$s<=$count;$s++)
                        {
                            echo"<br>".$s." : ".$stationID[$s].", ".$stationName[$s];
                        }
                        $obj = new Branch($objectCount, $lineIDwanted,
                            $count,$stationID,$stationName);
                        Branch::$branchObj[$objectCount] = $obj;
                        $finished=true; $count=0;
                    }
                }
            }
        }
    }
    return $objectCount;
}
?>

```

The method creates a Branch object for each linked list found in the **RoutePoint** table. The structure of the object is:

Attribute	Data type	Description
branchID	integer	identification number for the branch
lineID	integer	identification number for the underground line
stationID	array of integers	sequence of <b>stationID</b> values along the branch
stationName	array of strings	corresponding sequence of <b>station names</b> along the branch
pointCount	integer	number of entries in the stationID array

The station names are obtained from the stationID numbers by means of a **getStationName( )** method. Add this to the **Branch.php** file. Save the file and copy it to the server.

```
private static function getStationName($IDwanted,$stationCount)
{
    $name="";
    for ($i=1;$i<=$stationCount;$i++ )
    {
        if (Stations::$stationObj[$i]->stationID==$IDwanted)
            $name=Stations::$stationObj[$i]->stationName;
    }
    $name=str_replace("*"," ",$name);
    $name=trim($name);
    return $name;
}
}
?>
```

'Echo' text output lines were included in **loadBranch( )** to allow testing of the method. We can carry out these tests now. Return to the **findRoute.php** file and add a line of code near the start to include the **Branch** class:

```
<?
    $fromStation=$_REQUEST['fromStation'];
    $toStation=$_REQUEST['toStation'];
    include('RoutePoint.php');
    include('Line.php');
    include ('Stations.php');
    $routeResultCount=0;
    $routeResult=array();
    $routeResult[] = array();
    include ('Branch.php');
?>
<html>
```

Locate the block of code in **findRoute.php** which operates when a journey connection is found without a change of underground line. Add program lines to create Branch objects.

```

if (($fromFound==true)&&($toFound==true))
{
    $direct=true;
    $lineName=Line::getLineName($lineID);
    $routeResultCount++;
    $routeResult[$routeResultCount][1]=$fromStation;
    $routeResult[$routeResultCount][2]=$toStation;
    $routeResult[$routeResultCount][3]=$lineName;
    $branchCount=Branch::loadBranch($lineID);
}
}

```

Save **findRoute.php** and copy it to the server.

Run the web site and select a journey between two stations on the same underground line. When the 'find route' button is clicked, a list should be displayed which shows the stations in sequence along each of the branches of the selected line.

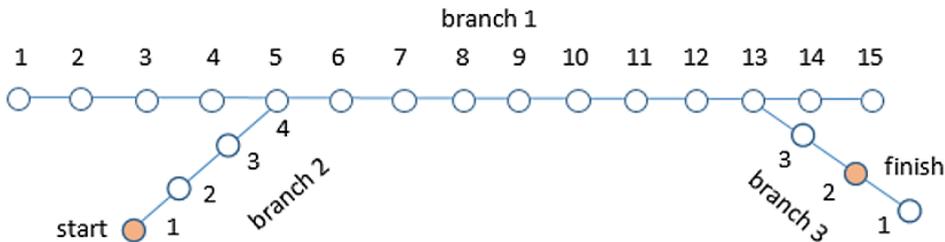
Belsize Park to Archway

Loading branches for line 2

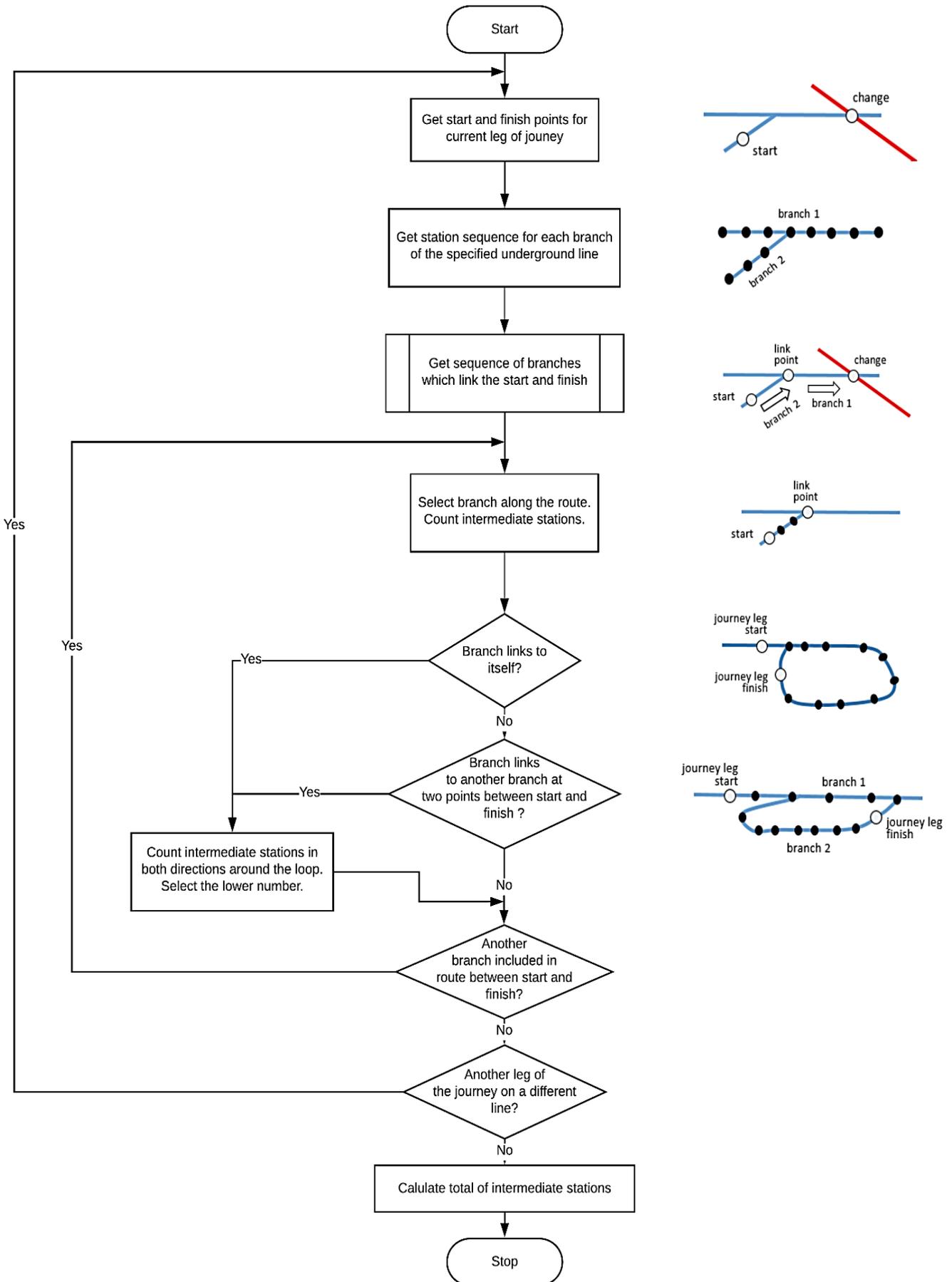
Branch 1	Branch 2	Branch 3
1: 197, High Barnet	1: 22, Edgware	1: 198, Mill Hill East
2: 243, Totteridge and Whetstone	2: 157, Burnt Oak	2: 193, Finchley Central
3: 244, Woodside Park	3: 156, Colindale	
4: 201, West Finchley	4: 154, Hendon Central	
5: 193, Finchley Central	5: 155, Brent Cross	
6: 239, East Finchley	6: 6, Golders Green	
.....	.....	

In some cases, **stationIDs** are shown with no station names alongside. These are the additional points inserted to allow the line to follow the map more accurately.

The next step is to calculate the number of intermediate stations between the start and destination stations for a journey along one underground line. The procedure can be extended later to allow journeys where a change of line is necessary. Although only a single underground line will be involved, the start and finish stations may lie on different branches of the line as in the example below.



A strategy for calculating the number of intermediate stations is outlined in the flowchart on the next page.



- The **loadBranch( )** method will be used to create lists of stations in sequence along each of the branches of the required underground line.
- If the start and destination lie on a single branch, the number of intermediate stations can be counted directly.
- If the start and destination lie on different branches, the number of stations which the route passes through on each branch are added to obtain the total number of intermediate stations for the complete journey.
- In the special cases of journeys involving loops of track, the numbers of intermediate stations will be calculated for both directions of travel around the loop. The smaller of these results will then be selected.

The calculations will be coordinated by a **countStations( )** method in the **Branch** class. This will take the **lineID** of an underground line and the **stationIDs** of two points on that line, then calculate and return the number of intermediate stations between these points.

- For a route where the start and destination stations lie on the same underground line, though not necessarily on the same branch, the **countStations( )** method will be called once.
- For a route involving one change, the **countStations( )** method will be called to find the number of stations between the start and change point on the first line. It will be called again to find the number of stations between the change point and destination on the second line, then the results will be added to obtain the complete journey.
- For a route involving two changes, the **countStations( )** method will be called three times for the different legs of the journey.

Return to the **Branch.php** class file and add the **locateStations( )** method.

```

public static function locateStations($fromStation,$toStation,$branchCount)
{
    for ($i=1;$i<=$branchCount;$i++)
    {
        $pointCount=Branch::$branchObj[$i]->pointCount;
        $location[$i][0]=0;
        $location[$i][1]=0;
        for ($j=1;$j<=$pointCount;$j++)
        {
            $currentStation = Branch::$branchObj[$i]->stationName[$j];
            if($currentStation == $fromStation)
            {
                $location[$i][0]= Branch::$branchObj[$i]->stationID[$j];
            }
            if($currentStation == $toStation)
            {
                $location[$i][1]= Branch::$branchObj[$i]->stationID[$j];
            }
        }
    }
    return $location;
}
}
?>

```

Continuing to work in the **Branch.php** class file, create the **countStations( )** method as shown below. This begins by calling the **locateStations( )** method to determine the numbers of the branches on which the start and finish stations lie. Save the **Branch.php** file and copy it to the server.

```

public static function countStations($fromStation,$toStation,$branchCount)
{
    $location = Branch::locateStations($fromStation,$toStation,$branchCount);
    for ($i=1;$i<=$branchCount;$i++)
    {
        echo"<br>Branch ".$i.".": From location = ".$location[$i][0]
        .": To location = ".$location[$i][1];
    }
}
?>

```

Return to the **findRoute.php** file and again locate the block of code which operates for a journey on a single underground line. Add a line to the program to call the **countStations( )** method.

```

if (($fromFound==true)&&($toFound==true))
{
    $direct=true;
    $lineName=Line::getLineName($lineID);
    $routeResultCount++;
    $routeResult[$routeResultCount][1]=$fromStation;
    $routeResult[$routeResultCount][2]=$toStation;
    $routeResult[$routeResultCount][3]=$lineName;
    $branchCount=Branch::loadBranch($lineID);

    $intermediate =
        Branch::countStations($fromStation,$toStation,$branchCount,$lineName);
}
}

```

Save **findRoute.php** and copy it to the server. Run the website and select a journey involving only one underground line, but beginning and ending on different branches. Click the 'find route' button and examine the output, as in the example below.

```

Woolwich Arsenal to Tower Gateway

Loading branches for line 8

Branch 1
1: 86, Bank
2: 459,
3: 460,
4: 89, Shadwell
5: 10, Limehouse
.....

Branch 2
1: 48, Lewisham
2: 146, Elverson Road
3: 145, Deptford Bridge
4: 57, Greenwich
5: 11, Cutty Sark for Maritime Greenwich
6: 19, Island Gardens
.....

Branch 3
1: 42, Woolwich Arsenal
2: 112, King George V
3: 18, London City Airport
4: 141, Pontoon Dock
5: 176, West Silverton
.....

Branch 4
1: 88, Tower Gateway
2: 460,

Branch 1: From location = 0: To location = 0
Branch 2: From location = 0: To location = 0
Branch 3: From location = 42: To location = 0
Branch 4: From location = 0: To location = 88

```

The stations should be shown in sequence for each branch as previously. A further block of text indicates if the required **stationID** values have been found on any of these branches, with results of 0 shown if not present.

Test the program for several journeys on different underground lines, but each time beginning and ending the journey on the same line. Check that the correct start branch and destination branch are selected, and that the correct stationID values are shown.

If stations are located correctly, the 'echo' lines in the **loadBranch( )** method in the **Branch.php** file can now be removed. Alternatively, the output lines can be de-activated by inserting `//` symbols at the start of the line, e.g.

```
public static function loadBranch($lineIDwanted)
{
    //echo"<p>Loading branches for line ".$lineIDwanted;
    $pointCount = RoutePoint::loadPoints();
```

We can now begin the calculation of intermediate stations during journeys. The results will be displayed as another column in the summary table.

Go to the **findRoute.php** file and locate the block of code which produces the table. Add lines of code as shown below, save the file and copy it to the server.

```
echo"<th>Second change</th>";
echo"<th>Third line</th>";
echo"<th>Map pixels</th>";

echo"<th>Intermediate stations</th></tr>";
$routeResult=compareDistances($routeResult,$routeResultCount);
for ($i=1;$i<=$routeResultCount;$i++)
{
    if (($resultCount<=4)||($routeResult[$i][8]>0))
    {
        echo"<tr><td>".$i."</td>";
        echo"<td>".$routeResult[$i][1]."</td>";
        echo"<td>".$routeResult[$i][2]."</td>";
        echo"<td>".$routeResult[$i][3]."</td>";
        echo"<td>".$routeResult[$i][4]."</td>";
        echo"<td>".$routeResult[$i][5]."</td>";
        echo"<td>".$routeResult[$i][6]."</td>";
        echo"<td>".$routeResult[$i][7]."</td>";
        echo"<td>".$routeResult[$i][8]."</td>";

        echo"<td>".$routeResult[$i][9]."</td>";

    }
}
echo"</table>";
```

Return to the **Branch.php** class file and locate the **countStations( )** method. Add the lines of program code shown below.

The program has a loop which checks each of the branches of the specified underground line. If the branch includes the stationIDs for both the start and destination, a **stationsBetween( )** method is called to determine the number of intermediate stations for the journey.

```

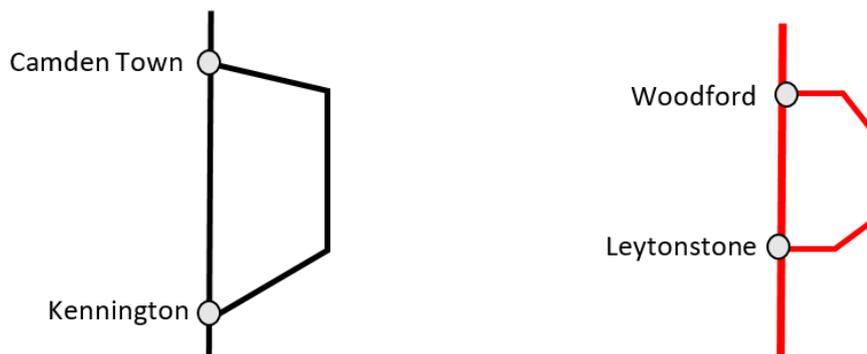
public static function countStations($fromStation,$toStation,$branchCount)
{
    $location = Branch::locateStations($fromStation,$toStation,$branchCount);
    for ($i=1;$i<=$branchCount;$i++)
    {
        echo"<br>Branch ".$i.".": From location = ".$location[$i][0]
            .": To location = ".$location[$i][1];

        $first=true;
        $foundRoute=false;
        for ($i=1;$i<=$branchCount;$i++)
        {
            if (($location[$i][0]>0)&&($location[$i][1]>0))
            {
                $foundRoute=true;
                $stations=Branch::stationsBetween($location[$i][0],
                    $location[$i][1],$i);
                if ($first==true)
                {
                    $intermediate = $stations;
                    $first = false;
                }
                else
                {
                    if ($stations<$intermediate)
                        $intermediate = $stations;
                }
            }
        }

        return $intermediate;
    }
}

```

We allow for the start and destination both occurring together on more than one branch, as in the case of journeys between Kennington and Camden Town on the Northern line, or Leytonstone and Woodford on the Central line, as shown below. In this case, calculations are carried out for both branches and the lower number of intermediate stations is selected.



Continuing to work in the **Branch.php** class file, add the **stationsBetween()** method on the next page. Save the **Branch.php** file and copy it to the server.

```

public static function stationsBetween($startStationID,$finishStationID,
                                     $branchID)
{
    $pointCount=Branch::$branchObj[$branchID]->pointCount;
    $fromFound=false;
    $toFound=false;
    $intermediate=0;
    $first=false;
    echo"<br>";
    for ($j=1;$j<=$pointCount;$j++)
    {
        $stationID = Branch::$branchObj[$branchID]->stationID[$j];
        $stationName = Branch::$branchObj[$branchID]->stationName[$j];
        echo"<br>".$j.": ".$stationID.", ".$stationName;
        if (($stationID==$finishStationID)||($stationID==$startStationID))
        {
            if ($fromFound==false)
            {
                $fromFound=true;
                $first=true;
                echo" *****from station";
            }
            else
            {
                $toFound=true;
                echo" *****to station";
            }
        }
        if (($fromFound==true)&&($toFound==false)&&($first==false))
        {
            if (strlen($stationName)>1)
            {
                $intermediate++;
                echo" [ ".$intermediate." ]";
            }
        }
        $first=false;
    }
    echo"<br>";
    return $intermediate;
}
}
?>

```

Go to **findRoute.php** and locate the block of code which called the **countStations( )** method. Add a line of code to copy the number of intermediate stations into the results array, ready to display in the summary table. Save the file and copy it to the server.

```

$routeResult[$routeResultCount][2]=$toStation;
$routeResult[$routeResultCount][3]=$lineName;
$branchCount=Branch::loadBranch($lineID);
$intermediate = Branch::countStations($fromStation,$toStation,$branchCount,$lineName);
$routeResult[$routeResultCount][9]=$intermediate;
}
}

```

Run the web site. Select a journey which can be completed on a single branch of one underground line. Click the 'find route' button and examine the output.

The **stationIDs** and **stationNames** should be listed in sequence along the branch. Start and finish stations for the journey are indicated. The direction of travel is not important, as we will simply count the number of intermediate stations for the journey. Note that route points with no station name are not included in the count.

The final result should be displayed as an additional column in the summary table.

```

Stockwell to Oxford Circus
Branch 1: From location = 23: To location = 2

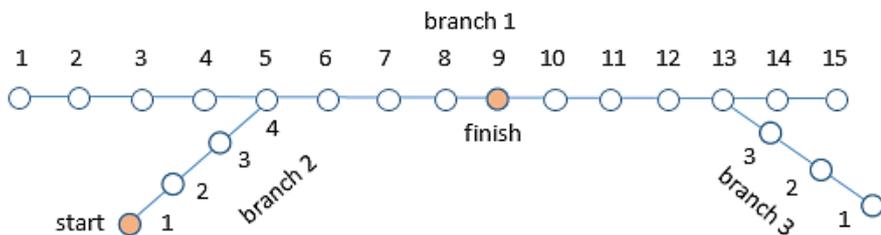
1: 7, Brixton
2: 23, Stockwell *****from station
3: 9, Vauxhall [1]
4: 455,
5: 46, Pimlico [2]
6: 374, Victoria [3]
7: 96, Green Park [4]
8: 451,
9: 2, Oxford Circus *****to station
10: 104, Warren Street
11: 106, Euston
12: 43, King's Cross St Pancras
    
```

Check that the correct numbers of intermediate stations are found for journeys involving only a single branch of any of the underground lines. Correct counts should be shown, with the possible exception of some journeys on the Circle line where the program has counted the 'long way' following the loop in the wrong direction as shown below. We will return to the problem of the Circle line later.

Edgware Road to King's Cross St Pancras

	From station	To station	First line	First change	Second line	Second change	Third line	Map pixels	Intermediate
1	Edgware Road	King's Cross St Pancras	Circle						22
2	Edgware Road	King's Cross St Pancras	Hammersmith and City						3

We now move on to consider start and destination stations lying on the same underground line, but on different branches. Let us assume that the start and destination branches are directly linked, as in the example below:



The total number of intermediate stations can be found by counting the stations along the route from the start station on branch 2 to the finish station on branch 1.

Return to the **countStations()** method in the **Branch.php** class file. Add a block of code as shown below which will call a **checkLink()** method if a direct route is not found between the start and destination on a single branch of the underground line.

```

        $stations=Branch::stationsBetween($location[$i][0], $location[$i][1],$i);
        if ($first==true)
        {
            $intermediate = $stations;
            $first = false;
        }
        else
        {
            if ($stations<$intermediate)
                $intermediate = $stations;
        }
    }
}
}

if ($foundRoute==false)
{
    $intermediate=Branch::checkLink($fromStation,$toStation,
                                    $branchCount,$location);
}

return $intermediate;
}

```

Add the **checkLink( )** method to the **Branch.php** class file, as shown below. Save the file and copy it to the server.

```

public static function checkLink($fromStation,$toStation,$branchCount,$location)
{
    $intermediate = 0;
    for ($i=1;$i<=$branchCount;$i++)
    {
        if ($location[$i][0]>0)
        {
            echo"<p>Start station on branch ".$i;
            $fromArray = Branch::$branchObj[$i]->stationID;
            $fromCount = Branch::$branchObj[$i]->pointCount;
            $fromBranch=$i;
        }
    }
    for ($i=1;$i<=$branchCount;$i++)
    {
        if ($location[$i][1]>0)
        {
            echo"<p>Finish station on branch ".$i;
            $toArray = Branch::$branchObj[$i]->stationID;
            $toCount = Branch::$branchObj[$i]->pointCount;
            $toBranch=$i;
        }
    }
    return $intermediate;
}
}
?>

```

Run the web site. Select a journey along a single underground line, but starting and finishing on different branches. Click the 'find route' button and examine the text output. The **checkLink( )** method identifies the branches for the start and destination stations, and creates two arrays called **fromArray** and **toArray** which contain the sequence of stationIDs along these branches.

```

Watford to Amersham
Branch 1: From location = 0: To location = 0
Branch 2: From location = 0: To location = 71
Branch 3: From location = 0: To location = 0
Branch 4: From location = 131: To location = 0

Start station on branch 4

Finish station on branch 2

```

The **checkLink()** method identifies the branches for the start and destination stations, and creates two arrays called **fromArray** and **toArray** which contain the sequence of stationIDs along these branches.

The next step is to compare the arrays to determine whether one or more station points occur in both. If so, a connection between the branches has been found. Return to the **checkLink()** method in the **Branch.php** file and add lines of program code to call a **compareArrays()** method.

```

        $toCount = Branch::$branchObj[$i]->pointCount;
        $toBranch=$i;
    }
}
$resultArray = Branch::compareArrays($fromArray,$fromCount,$toArray,$toCount);
$resultCount=count($resultArray);
return $intermediate;
}

```

Add the **compareArrays()** method at the end of the **Branch.php** file, as shown below

```

private static function compareArrays($array1,$count1,$array2,$count2)
{
    $stationCount=Stations::loadStations();
    echo"<p>Array 1";
    $resultCount=0;
    for ($k=1;$k<=$count1;$k++)
        echo"<br>".$k."": ".$array1[$k].", ".Branch::getStationName($array1[$k],
                                                                    $stationCount);

    echo"<p>Array 2";
    for ($k=1;$k<=$count2;$k++)
        echo"<br>".$k."": ".$array2[$k].", ".Branch::getStationName($array2[$k],
                                                                    $stationCount);

    for ($i=1;$i<=$count1;$i++)
    {
        for ($j=1;$j<=$count2;$j++)
        {
            if ($array1[$i]==$array2[$j])
            {
                $resultCount++;
                $resultArray[$resultCount]=$array1[$i];
            }
        }
    }
    echo"<p>Link points between branches";
    for ($k=1;$k<=$resultCount;$k++)
        echo"<br>".$k."": ".$resultArray[$k].", "
        .Branch::getStationName($resultArray[$k], $stationCount);
    return $resultArray;
}
}

```

Save the **Branch.php** file and copy it to the server. Run the web site. Select a journey along a single underground line, where the start and destination stations lie on two connected branches. Click the 'find route' button and examine the program output. Stations are listed for the start and destination branches and any shared stations are identified as link points, as in this example.

```

Ealing Broadway to Perivale
Branch 1: From location = 120: To location = 0
Branch 2: From location = 0: To location = 0
Branch 3: From location = 0: To location = 235

Start station on branch 1
Finish station on branch 3

Array 1
1: 120, Ealing Broadway
2: 205, West Acton
3: 477,
4: 204, North Acton
5: 234, East Acton
6: 295, White City

Array 2
1: 477,
2: 121, Hanger Lane
3: 235, Perivale
4: 320, Greenford
5: 473,
6: 311, Northolt
7: 161, South Ruislip
8: 312, Ruislip Gardens
9: 159, West Ruislip

Link points between branches
1: 477,
    
```

Return to the **Branch.php** file and add another method as shown below. This takes the name of a station and obtains the equivalent stationID from the array within the branch object.

```

private static function getStationID($stationWanted,$branch)
{
    $stationID=0;
    $pointCount = Branch::$branchObj[$branch]->pointCount;
    for ($i=1;$i<=$pointCount;$i++ )
    {
        $currentStation = Branch::$branchObj[$branch]->stationName[$i];
        if ($currentStation==$stationWanted)
        {
            $stationID=Branch::$branchObj[$branch]->stationID[$i];
        }
    }
    return $stationID;
}
?>
    
```

We now have the components needed to calculate the number of intermediate stations for a journey between two connected branches of a single underground line. Return to the **checkLink( )** method in the **Branch.php** file and add the block of program code shown below.

Save the **Branch.php** file and copy it to the server. Run the web site and select a journey on a single underground line between two linked branches.

```

        $toCount = Branch::$branchObj[$i]->pointCount;
        $toBranch=$i;
    }
}
$resultArray = Branch::compareArrays($fromArray,$fromCount,$toArray,$toCount);
$resultCount=count($resultArray);

$fromStationID = Branch::getStationID($fromStation,$fromBranch);
$toStationID = Branch::getStationID($toStation,$toBranch);
$stationCount=Stations::loadStations();
$first=true;
if ($resultCount>0)
{
    for ($k=1;$k<=$resultCount;$k++)
    {
        $intermediate1 = Branch::stationsBetween($fromStationID,
                                                $resultArray[$k],$fromBranch);
        $intermediate2 = Branch::stationsBetween($resultArray[$k],
                                                $toStationID,$toBranch);

        $total = $intermediate1 + $intermediate2;
        $linkPointName=Branch::getStationName($resultArray[$k],$stationCount);
        if (strlen($linkPointName)>2)
            $total++;
        if ($first==true)
        {
            $first=false;
            $intermediate=$total;
        }
        else
        {
            if ($total<$intermediate)
                $intermediate=$total;
        }
    }
}
else
{
    echo"<p>Checking for connections between three branches...";
}

return $intermediate;
}

```

The program lists the stations in each branch, and also identifies the start and finish points for the journey segments on each of the branches. The numbers of intermediate stations are recorded, and the total displayed in the results table.

```

1: 131, Watford *****from station
2: 317, Croxley [1]
3: 465, *****to station

```

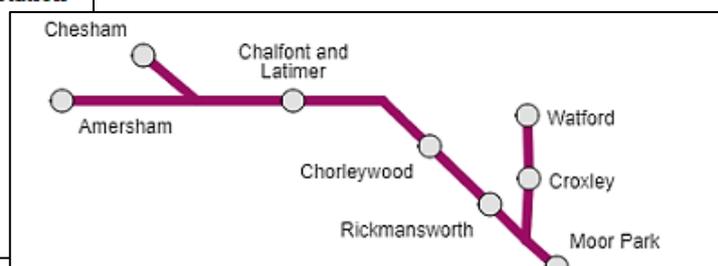
```

1: 71, Amersham *****from station
2: 463,
3: 314, Chalfont and Latimer [1]
4: 464,
5: 318, Chorleywood [2]
6: 315, Rickmansworth [3]
7: 465, *****to station
8: 309, Moor Park
9: 316, Northwood

```

#### Journey from Watford to Amersham

- 1 intermediate station found on the Watford branch.
- 3 intermediate stations found on the Amersham branch.



Return to the **checkLink()** method in the **Branch.php** file. Program code can now be added to find routes where the start and destination branches are linked via a third intermediate branch. Insert the block of code shown below and continued on the next page.

```

else
{
    echo"<p>Checking for connections between three branches...";

    for ($i=1;$i<=$branchCount;$i++)
    {
        if (($location[$i][0]==0)&&($location[$i][1]==0))
        {
            echo"<p>Checking for connection via branch ".$i;
            $linkArray = Branch::$branchObj[$i]->stationID;
            $linkCount = Branch::$branchObj[$i]->pointCount;
            $linkBranch=$i;
            $resultArray1 = Branch::compareArrays($fromArray,
                $fromCount,$linkArray,$linkCount);
            echo"<p>Returned RESULT fromArray to linkArray *****";
            $resultCount1=count($resultArray1);
            for ($k=1;$k<=$resultCount1;$k++)
            echo"<br>".$k." : ".$resultArray1[$k];
            $resultArray2 = Branch::compareArrays($toArray,
                $toCount,$linkArray,$linkCount);
            echo"<p>Returned RESULT toArray to linkArray *****";
            $resultCount2=count($resultArray2);
            for ($k=1;$k<=$resultCount2;$k++)
                echo"<br>".$k." : ".$resultArray2[$k];
            if (($resultCount1>0)&&($resultCount2>0))
            {
                echo"<p>Connection found *****";
                for ($a=1;$a<=$resultCount1;$a++)
                {
                    for ($b=1;$b<=$resultCount2;$b++)
                    {
                        echo"<br>Travel from ".$fromStation." on branch "
                            .$fromBranch." Link point ".$resultArray1[$a]
                            ." to branch ".$linkBranch." Link point ".$resultArray2[$b]
                            ." to branch ".$toBranch." Finish at ".$toStation;
                        $intermediate1 = Branch::stationsBetween($fromStationID,
                            $resultArray1[$a],$fromBranch);
                        $intermediate2 = Branch::stationsBetween($resultArray1[$a],
                            $resultArray2[$b],$linkBranch);
                        $intermediate3 = Branch::stationsBetween($toStationID,
                            $resultArray2[$b],$toBranch);
                        $total = $intermediate1 + $intermediate2 + $intermediate3;
                        $linkPointName1=Branch::getStationName($resultArray1[$a],
                            $stationCount);
                        if (strlen($linkPointName1)>2)
                            $total++;
                        $linkPointName2=Branch::getStationName($resultArray2[$b],
                            $stationCount);
                        if (strlen($linkPointName2)>2)
                            $total++;
                        if ($first==true)
                        {
                            $first=false;
                            $intermediate=$total;
                        }
                    }
                }
            }
        }
    }
}

```

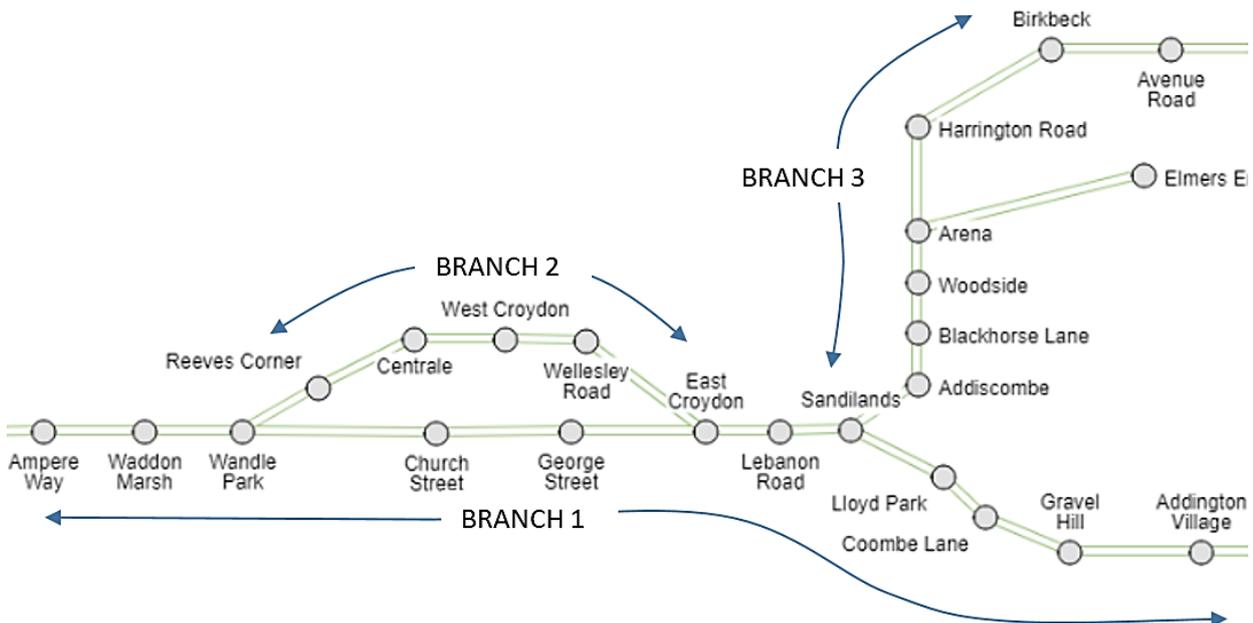
```

        if ($first==true)
        {
            $first=false;
            $intermediate=$total;
        }

        else
        {
            if ($total<$intermediate)
                $intermediate=$total;
        }
        echo"<p> Total intermediate stations = ".$intermediate."<br>";
    }
}
}
}
}
}
}
return $intermediate;
}
}

```

Save the **Branch.php** file and copy it to the server. Run the web site and select a journey involving travel from a start branch, via an intermediate branch, to a destination branch of the same underground line. An example would be the journey from West Croydon on branch 2 of the tram network, via branch 1, to the destination of Harrington Road on branch 3.



An extract from the program output is shown on the next page.

- The procedure begins by determining the branches on which the start and destination stations lie. It is already known that there is no direct connection between these branches.
- The program tests each of the remaining branches in turn as a possible link between the start and destination. If shared station points are found with both the start and destination branches, then a connection is possible.
- Each set of connecting station points is considered in turn. For the example above, the journey from West Croydon could be made by two different routes:
  - West Croydon - Wandle Park - Sandilands - Harrington Road
  - West Croydon - East Croydon - Sandilands - Harrington Road

- The two possible routes are considered in turn. Start and finish stations for each leg of the journey are identified, and the number of intermediate stations counted. A provisional total is obtained for the whole journey.
- The program checks whether the branches are linked at stations or at intermediate points along the line. If links occur at stations, then these are added to the total of intermediate stations.
- When calculations are completed, the lowest total is selected from the alternative routes. This result is displayed in the summary table.

Travel from West Croydon on branch 2. Link point 382 to branch 1 Link point 384 to branch 3.  
Finish at Harrington Road

1: 379, Wandle Park  
2: 407, Reeves Corner  
3: 406, Centrale  
4: 404, West Croydon \*\*\*\*\*from station  
5: 405, Wellesley Road [1]  
6: 382, East Croydon \*\*\*\*\*to station

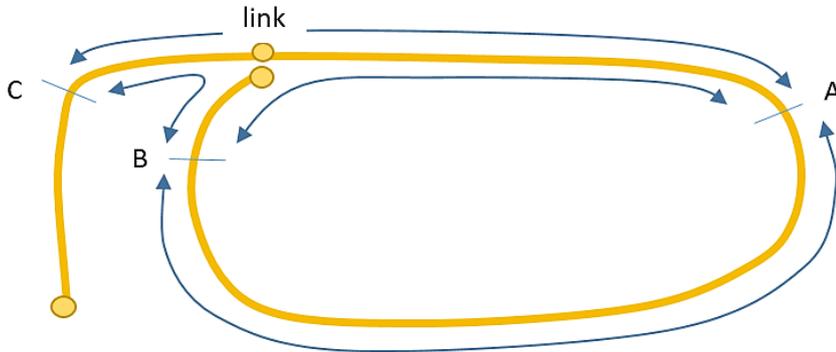
1: 369, Wimbledon  
2: 367, Dundonald Road  
3: 449,  
4: 368, Merton Park  
5: 450,  
6: 370, Morden Road  
7: 371, Phipps Bridge  
8: 372, Belgrave Walk  
9: 348, Mitcham  
10: 373, Mitcham Junction  
11: 375, Beddington Lane  
12: 376, Therapia Lane  
13: 377, Ampere Way  
14: 378, Waddon Marsh  
15: 379, Wandle Park  
16: 380, Church Street  
17: 381, George Street  
18: 382, East Croydon \*\*\*\*\*from station  
19: 383, Lebanon Road [1]  
20: 384, Sandilands \*\*\*\*\*to station  
21: 385, Lloyd Park  
22: 395, Coombe Lane  
23: 396, Gravel Hill  
24: 397, Addington Village  
25: 398, Fieldway  
26: 399, King Henry`s Drive  
27: 400, New Addington

1: 384, Sandilands \*\*\*\*\*from station  
2: 386, Addiscombe [1]  
3: 387, Blackhorse Lane [2]  
4: 388, Woodside [3]  
5: 389, Arena [4]  
6: 390, Harrington Road \*\*\*\*\*to station  
7: 391, Birkbeck  
8: 392, Avenue Road  
9: 393, Beckenham Road  
10: 394, Beckenham Junction

Total intermediate stations = 8

Almost all journeys on the London underground network can be made via one, two or three branches of any line. It is left as a programming exercise to count the number of intermediate stations for a journey involving four branches of an underground line if required.

We now return to the problem identified earlier for some journeys on the Circle line. The line is topologically a loop with a tail.



- A journey from point A to point B could be made by travelling around the loop in either a clockwise or anti-clockwise direction. The route passing through the least number of intermediate stations would be chosen.
- A journey from point A or B to point C would first involve travelling around the loop to the link point, clockwise or anti-clockwise depending on which route passed through the least number of intermediate stations.

Consequently, most journeys on the Circle line require a choice to be made about the direction of travel around the loop. This choice will be made by a **circleCount()** method which we will add to the Branch class file.

Go to the **Branch.php** file and add lines of program code to the beginning of the **countStations()** method as shown below.

```
public static function countStations($fromStation,$toStation,$branchCount)
{
    $lineID = Branch::$branchObj[1]->lineID;
    $lineName = Line::getLineName($lineID);
    if ($lineName=='Circle')
    {
        echo"<p>Calculating for Circle line...";
        $intermediate=Branch::circleCount($fromStation,$toStation);
    }
    else
    {
        $location = Branch::locateStations($fromStation,$toStation,$branchCount);
        $first=true;
```

Add a further bracket to close the **else..** condition before the **return** line at the end of the method.

```
        if ($foundRoute==false)
        {
            $intermediate=Branch::checkLink($fromStation,$toStation,
                $branchCount,$location);
        }
    }
    return $intermediate;
}
```

Add the **circleCount( )** method at the end of the **Branch.php** file. Save the file and copy it to the server.

```

public static function circleCount($fromStation,$toStation)
{
    $count=Branch::$branchObj[1]->pointCount;
    for ($i=1;$i<=$count;$i++)
    {
        $stationID = Branch::$branchObj[1]->stationID[$i];
        $stationName = Branch::$branchObj[1]->stationName[$i];
        echo"<br>".$i." : ".$stationID.", ".$stationName;
    }
    return $intermediate;
}
?>

```

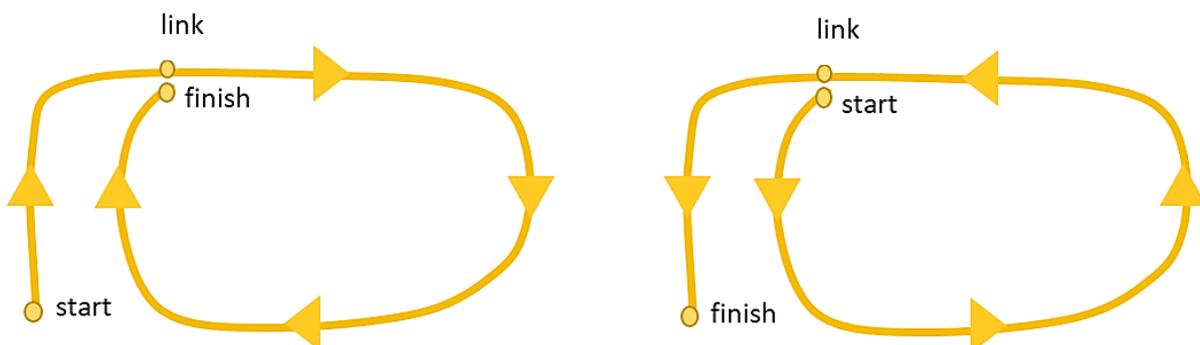
Run the web site and select a journey beginning and ending on the Circle line. Click the 'find route' button. The program will display a list of the stations in sequence along the Circle line. The order may be reversed from the example below, depending on the direction in which the line was entered.

```

Shepherd's Bush Market to High Street Kensington
Calculating for Circle line...
1: 481, Hammersmith
2: 297, Goldhawk Road
3: 290, Shepherd's Bush Market
4: 296, Wood Lane
5: 292, Latimer Road
6: 122, Ladbroke Grove
7: 293, Westbourne Park

```

A loop is created by either the first or last station connecting to a link station along the line.



The next step is to identify the link station. Return to the **circleCount( )** method in the **Branch.php** class file. Add lines of code as shown below. Save the file and copy it to the server.

The method begins by noting the names of the first and last stations along the line. As the loop prints each station name, a watch is kept for the first or last station name appearing again to create the Circle link.

```

public static function circleCount($fromStation,$toStation)
{
    $count=Branch::$branchObj[1]->pointCount;
    $first = Branch::$branchObj[1]->stationName[1];
    $last = Branch::$branchObj[1]->stationName[$count];

    for ($i=1;$i<=$count;$i++)
    {
        $stationID = Branch::$branchObj[1]->stationID[$i];
        $stationName = Branch::$branchObj[1]->stationName[$i];
        echo"<br>".$i.".": ".$stationID.", ".$stationName;

        if(($i>1)&&($i<$count))
        {
            if (($stationName==$first)||($stationName==$last))
            {
                echo" ***** link *****";
                $linkLocation = $i;
                $linkStationCount= $betweenStations;
                $loopFirst=true;
                if ($stationName==$last)
                {
                    $loopFirst=false;
                }
            }
        }
    }
    return $intermediate;
}

```

The method begins by noting the names of the first and last stations along the line. As the loop prints each station name, a watch is kept for the first or last station name appearing again to create the Circle link.

Run the web site. Again select a journey on the circle line. Check that the link station is identified in the station list.

7: 293, Westbourne Park	39: 52, High Street Kensington
8: 291, Royal Oak	40: 100, Notting Hill Gate
9: 475,	41: 211, Bayswater
10: 125, Paddington	42: 126, Paddington
11: 210, Edgware Road ***** link *****	43: 209, Edgware Road
12: 25, Baker Street	
13: 135, Great Portland Street	

last station in the list

Return to the **Branch.php** class file. Go to the start of the **circleCount ( )** method and add several variables.

```

public static function circleCount($fromStation,$toStation)
{
    $count=Branch::$branchObj[1]->pointCount;
    $first = Branch::$branchObj[1]->stationName[1];
    $last = Branch::$branchObj[1]->stationName[$count];

    $wanted=0;
    $betweenStations=0;
    $countingBetween=false;

    for ($i=1;$i<=$count;$i++)
    {

```

The next step is to identify the start and destination stations for a journey on the Circle line, and to count the number of stations between these points. Add the lines of program code to the `circleCount()` method.

```

        if ($stationName==$last)
        {
            $loopFirst=false;
        }
    }
}
}
if (($stationName==$fromStation)||($stationName==$toStation))
{
    echo " ***** STATION WANTED *****";
    $wanted++;
    $wantedLocation[$wanted] = $i;
    if ($wanted==1)
    {
        $countingBetween=true;
    }
}
if ($countingBetween==true)
{
    if (strlen($stationName)>2)
    {
        $betweenStations++;
        echo " [ ".$betweenStations." ]";
    }
}
if ($wanted==2)
{
    $countingBetween=false;
}
}
return $intermediate;
}

```

Save the **Branch.php** file and copy it to the server. Run the web site and select a journey on the Circle line. Click the 'find route' button. The list of stations is again displayed, but the start and destination are identified, as shown in the example below. A count is kept of the number of stations from the start to the destination, as displayed in square brackets in the screen output.

**Shepherd's Bush Market to High Street Kensington**

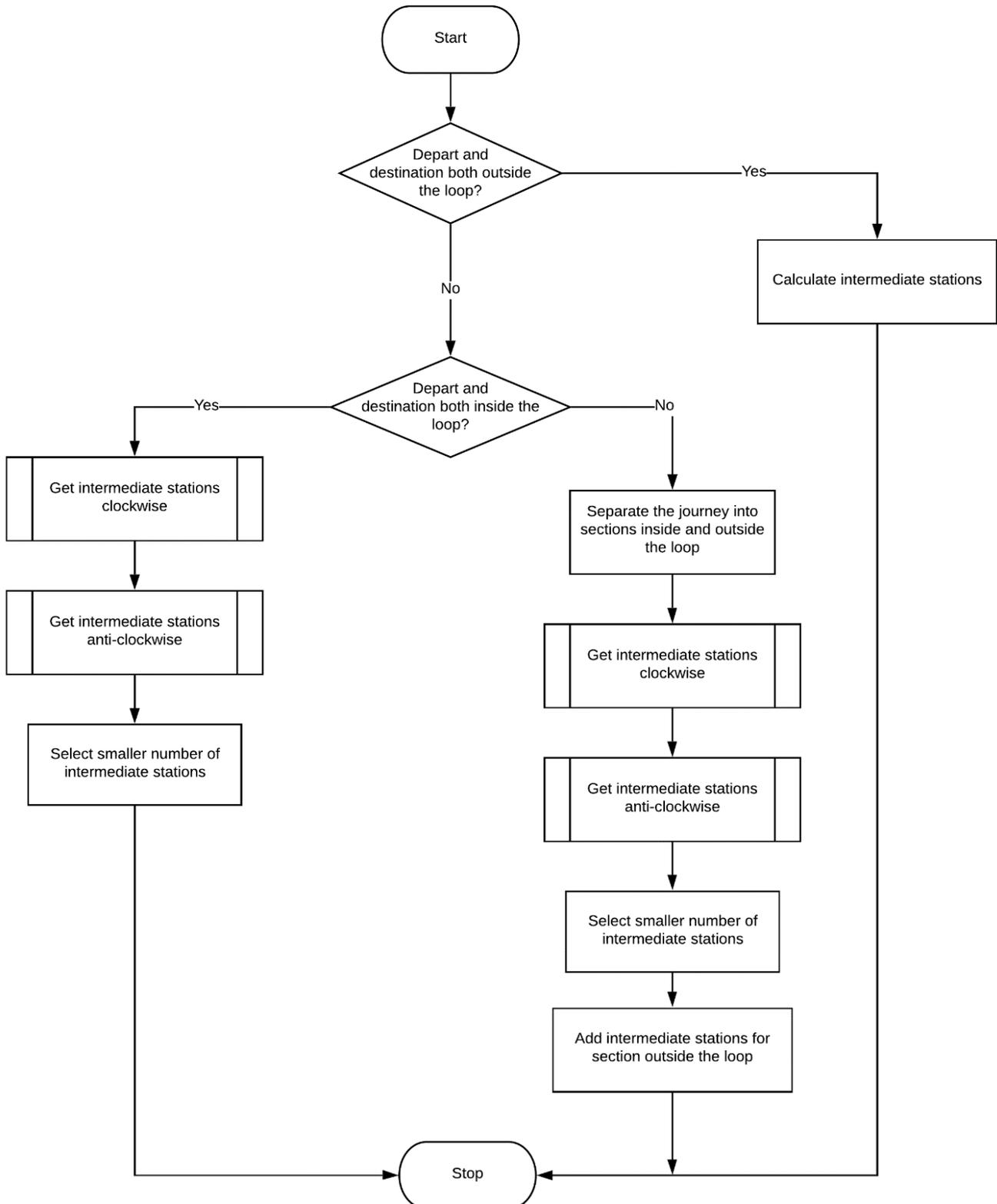
Calculating for Circle line....

- 1: 481, Hammersmith
- 2: 297, Goldhawk Road
- 3: 290, Shepherd's Bush Market \*\*\*\*\* STATION WANTED \*\*\*\*\* [1]
- 4: 296, Wood Lane [2]
- 5: 292, Latimer Road [3]
- 6: 122, Ladbroke Grove [4]
- 7: 293, Westbourne Park [5]
- 8: 291, Royal Oak [6]
- 9: 475,
- 10: 125, Paddington [7]
- 11: 210, Edgware Road \*\*\*\*\* link \*\*\*\*\* [8]
- 12: 25, Baker Street [9]

- 35: 16, Sloane Square [28]
- 36: 55, South Kensington [29]
- 37: 268, Gloucester Road [30]
- 38: 484,
- 39: 52, High Street Kensington \*\*\*\*\* STATION WANTED \*\*\*\*\* [31]
- 40: 100, Notting Hill Gate
- 41: 211, Bayswater
- 42: 126, Paddington
- 43: 209, Edgware Road

We now have all the data necessary for calculating a shortest journey route on the Circle line. The procedure is illustrated in the flowchart below.

For the simple case of the journey starting and finishing on the section of line outside the loop, the number of intermediate stations has already been calculated. For journeys which occur partly or completely within the Circle loop, we must find the journey distances in both the clockwise and anti-clockwise map directions and choose the shortest of these.



Return to the **circleCount( )** method. Add lines of program code which will count the number of stations around the Circle loop, ignoring non-station route points which were added to make the map display more accurate.

```

        if ($wanted==2)
        {
            $countingBetween=false;
        }
    }
    $loopStations=0;
    $countingLoop=false;
    if ($loopFirst==true)
    {
        $a=1;
        $b=$linkLocation;
    }
    else
    {
        $a=$linkLocation;
        $b=$count;
    }
    for ($i=$a;$i<=$b;$i++)
    {
        $stationName = Branch::$branchObj[1]->stationName[$i];
        if (strlen($stationName)>2)
        {
            $loopStations++;
        }
    }
    echo"<p>Loop station count = ".$loopStations;
    return $intermediate;
}

```

Save the **Branch.php** file and copy it to the server. Run the web site and select a journey on the Circle line. Click the 'find route' button. The number of stations around the Circle loop should be found and displayed. Check that the count is correct.

```

41: 211, Bayswater
42: 126, Paddington
43: 209, Edgware Road

Loop station count = 28

```

The next stage is to split a selected journey into sections inside and outside the Circle loop. This can be done because we earlier recorded the position of the link station if it occurred between the start and finish of a journey along the Circle line.

Return to the **circleCount( )** method and add the lines of program code below.

```

}
echo"<p>Loop station count = ".$loopStations;

$insideLoop=0;
$outsideLoop=0;
if ($loopFirst==true)
{
  if ($linkStationCount==0)
    $outsideLoop=$betweenStations;
  else
    if ($linkStationCount==$betweenStations)
      $insideLoop=$betweenStations;
    else
    {
      $insideLoop=$linkStationCount+1;
      $outsideLoop=$betweenStations-$linkStationCount;
    }
}
if ($loopFirst==false)
{
  if ($linkStationCount==0)
    $insideLoop=$betweenStations;
  else
    if ($linkStationCount==$betweenStations)
      $outsideLoop=$betweenStations;
    else
    {
      $outsideLoop=$linkStationCount+1;
      $insideLoop=$betweenStations-$linkStationCount;
    }
}
echo"<p>Outside loop = ".$outsideLoop;
echo"<p>Inside loop = ".$insideLoop;

return $intermediate;
}

```

Save the **Branch.php** file and copy it to the server. Select a journey on the Circle line which occurs partly inside and partly outside the Circle loop, such as from Ladbroke Grove to Euston Square.



Click the 'find route' button and examine the program output. The route has been split into two sections. Counts have been made to the link station of Edgware Road. The results are Inclusive of the end stations of each journey section. Check that the values are correct.

Ladbroke Grove to Euston Square

Outside loop = 5

Inside loop = 4

The final step is to calculate the number of stations for travel in the opposite direction within the Circle loop. The smaller number of intermediate stations can then be selected. Any additional stations for a section of journey outside the loop can then be added.

```

    echo"<p>Outside loop = ".$outsideLoop;
    echo"<p>Inside loop = ".$insideLoop;

    if ($insideLoop==0)
        $intermediate=$outsideLoop-2;
    else
    {
        $intermediate1=$insideLoop-2;
        $intermediate2=($loopStations-$insideLoop);
        if ($intermediate1<$intermediate2)
            $intermediate=$intermediate1;
        else
            $intermediate=$intermediate2-1;
        if ($outsideLoop>=2)
            $intermediate=$intermediate+($outsideLoop-1);
    }
    return $intermediate;
}
    
```

Save the **Branch.php** file and copy it to the server. Select a variety of journeys on the Circle line, such as the journey from Latimer Road to Notting Hill Gate in this example.



In each case, check that the number of intermediate stations is displayed correctly in the summary table. The calculation assumes that a journey joining the Circle loop may then continue in either a clockwise or anti-clockwise map direction.

From station	To station	First line	First change	Second line	Second change	Third line	Map pixels	Intermediate
Latimer Road	Notting Hill Gate	Circle						7

Return to the **Branch.php** file. All 'echo' lines can now be removed or de-activated, so that just the results table is output. Save the **Branch.php** file and copy it to the server.

Go now to the **findRoute.php** file and identify the section of the program where the results table is output. We will extend the program to output the total numbers of intermediate stations for journeys involving one change of underground line. This will be done by calling the **countStations( )** method for each separate leg of the journey, then adding the numbers of intermediate stations. One additional station will be added to the total, to include the station where the traveller changes underground line.

```

echo"<td>".$routeResult[$i][6]."</td>";
echo"<td>".$routeResult[$i][7]."</td>";
echo"<td>".$routeResult[$i][8]."</td>";

if ($direct==false)
{
    $lineID= Line::getIDfromName($routeResult[$i][3]);
    $lineName=Line::getLineName($lineID);
    $branchCount=Branch::loadBranch($lineID);
    $firstCount = Branch::countStations($routeResult[$i][1],
                                        $routeResult[$i][4],$branchCount,$lineName);
    $lineID= Line::getIDfromName($routeResult[$i][5]);
    $lineName=Line::getLineName($lineID);
    $branchCount=Branch::loadBranch($lineID);
    $secondCount = Branch::countStations($routeResult[$i][4],
                                        $routeResult[$i][2],$branchCount,$lineName);
    $total = $firstCount+$secondCount+1;
    $routeResult[$i][9]=$total;
}

echo"<td>".$routeResult[$i][9]."</td>";
}
}
echo"</table>";

```

Save the **findRoute.php** file and copy it to the server.

Run the web site. Select various journeys requiring one change of underground line, such as the journey from Notting Hill Gate to Russell Square in the example below. Check that intermediate stations have been calculated correctly for the route options displayed in the results table.

From station	To station	First line	First change	Second line	Second change	Third line	Map pixels	Intermediate
Notting Hill Gate	Russell Square	Central	Holborn	Piccadilly			499	7
Notting Hill Gate	Russell Square	Circle	South Kensington	Piccadilly			610	10
Notting Hill Gate	Russell Square	Circle	King's Cross St Pancras	Piccadilly			605	7
Notting Hill Gate	Russell Square	District	South Kensington	Piccadilly			610	10

Return to the **findRoute.php** file and add lines of program code to calculate the numbers of intermediate stations for journeys involving two changes of underground line, as shown on the next page.

The **countStations( )** method is now called three times for the three legs of the journey.

Save the **findRoute.php** file and copy it to the server.

Run the web site and select a variety of longer journeys requiring two changes of underground line. In each case, check that correct totals of intermediate stations are given for each route option.

	From station	To station	First line	First change	Second line	Second change	Third line	Map pixels	Intermediate
1	Maida Vale	Pontoon Dock	Bakerloo	Oxford Circus	Central	Bank	DLR	1554	20
9	Maida Vale	Pontoon Dock	Bakerloo	Baker Street	Jubilee	Canning Town	DLR	1566	17
10	Maida Vale	Pontoon Dock	Bakerloo	Baker Street	Jubilee	Canary Wharf	DLR	1565	28
17	Maida Vale	Pontoon Dock	Bakerloo	Charing Cross	Northern	Bank	DLR	1608	24

```

if ($direct==false)
{
    $lineID= Line::getIDfromName($routeResult[$i][3]);
    $lineName=Line::getLineName($lineID);
    $branchCount=Branch::loadBranch($lineID);
    $firstCount = Branch::countStations($routeResult[$i][1],
        $routeResult[$i][4],$branchCount,$lineName);
    $lineID= Line::getIDfromName($routeResult[$i][5]);
    $lineName=Line::getLineName($lineID);
    $branchCount=Branch::loadBranch($lineID);

    if (strlen($routeResult[$i][7])<1)
    {
        $secondCount = Branch::countStations($routeResult[$i][4],
            $routeResult[$i][2],$branchCount,$lineName);
        $total = $firstCount+$secondCount+1;
        $routeResult[$i][9]=$total;
    }
    else
    {
        $secondCount = Branch::countStations($routeResult[$i][4],
            $routeResult[$i][6],$branchCount,$lineName);
        $lineID= Line::getIDfromName($routeResult[$i][7]);
        $lineName=Line::getLineName($lineID);
        $branchCount=Branch::loadBranch($lineID);
        $thirdCount = Branch::countStations($routeResult[$i][6],
            $routeResult[$i][2],$branchCount,$lineName);
        $total = $firstCount+$secondCount+$thirdCount+2;
        $routeResult[$i][9]=$total;
    }
    }
    echo"<td>".$routeResult[$i][9]."</td>";
}
}
echo"</table>";

```

We can now move ahead to find shortest journey option, based on the number of intermediate stations. Return to the **findRoute.php** file and add lines of program code as shown in the two boxes below.

```

    echo"<td>".$routeResult[$i][9]."</td>";
}
}
echo"</table>";

$first=true;
for ($i=1;$i<=$routeResultCount;$i++)
{
    if (($resultCount<=4)||($routeResult[$i][8]>0))
    {
        if ($first==true)
        {
            $min=$i;
            $minStations=$routeResult[$i][9];
            $first=false;
        }
        else
        {

```

```

        else
        {
            if ($routeResult[$i][9]<$minStations)
            {
                $min=$i;
                $minStations=$routeResult[$i][9];
            }
        }
    }
}

echo"<form method=post action='index.php?message=" . $message . "'>";
echo"<p><input type=submit value='continue'>";
echo"</form>";
?>
</body>
</html>

```

This block of code selects the **routeResult[ ]** array value with the lowest number of intermediate stations recorded.

We now use data from the **routeResult[ ]** array to create a text string **message2** which will output the preferred route for the traveller. Add the lines of program code shown below, then save the **findRoute.php** file and copy it to the server.

```

        if ($routeResult[$i][9]<$minStations)
        {
            $min=$i;
            $minStations=$routeResult[$i][9];
        }
    }
}

if ($direct==true)
{
    $message2="Travel from ".$routeResult[$min][1]." to "
        . $routeResult[$min][2]." on the ".$routeResult[$min][3]." line.";
}
else
{
    $message2="Travel from ".$routeResult[$min][1]." to "
        . $routeResult[$min][4]." on the ".$routeResult[$min][3]." line.";
    if (strlen($routeResult[$min][7])<1)
    {
        $message2=$message2."<br>Travel from ".$routeResult[$min][4]." to "
            . $routeResult[$min][2]." on the ".$routeResult[$min][5]." line.";
    }
    else
    {
        $message2=$message2."<br>Travel from ".$routeResult[$min][4]." to "
            . $routeResult[$min][6]." on the ".$routeResult[$min][5]." line.";
        $message2=$message2."<br>Travel from ".$routeResult[$min][6]." to "
            . $routeResult[$min][2]." on the ".$routeResult[$min][7]." line.";
    }
}
echo"<p>Message 2: ".$message2;

echo"<form method=post action='index.php?message=" . $message . "'>";
echo"<p><input type=submit value='continue'>";
echo"</form>";

```

Run the web site. Carry out tests with a variety of journeys, either using a single underground line or requiring one or two changes of line. In each case, check that **message2** correctly describes the route option with the least number of intermediate stations.

	From station	To station	First line	First change	Second line	Second change	Third line	Map pixels	Intermediate
1	Knightsbridge	Oxford Circus	Piccadilly	Piccadilly Circus	Bakerloo			292	3
2	Knightsbridge	Oxford Circus	Piccadilly	Holborn	Central			508	7
3	Knightsbridge	Oxford Circus	Piccadilly	Green Park	Victoria			203	2
4	Knightsbridge	Oxford Circus	Piccadilly	King's Cross St Pancras	Victoria			696	10

Message 2: Travel from Knightsbridge to Green Park on the Piccadilly line.  
Travel from Green Park to Oxford Circus on the Victoria line.

Return to the **findRoute.php** file. Add a similar block of program code to select the journey option with the second-least number of intermediate stations. This block will only operate if more than one route option is present in the results table.

```

echo"<p>Message 2: ".$message2;
if ($routeResultCount>1)
{
    $routeResult[$min][8]= -1;
    $oldMin=$min;
    $first=true;
    for ($i=1;$i<=$routeResultCount;$i++)
    {
        if (($direct==true)||(($resultCount<=4)||($routeResult[$i][8]>0)))
        {
            if (($first==true)&&($i!=$oldMin))
            {
                $min=$i;
                $minStations=$routeResult[$i][9];
                $first=false;
            }
            else
            {
                if (($routeResult[$i][9]<$minStations)&&($i!=$oldMin))
                {
                    $min=$i;
                    $minStations=$routeResult[$i][9];
                }
            }
        }
    }
}

```

```

echo"<form method=post action='index.php?message=".$message."'>";
echo"<p><input type=submit value='continue'>";
echo"</form>";

```

Continuing to work in the **findRoute.php** file, add the block of program code below. This creates a string **message3** describing an alternative route which the traveller might take.

Save the **findRoute.php** file and copy it to the server.

```

        if ($routeResult[$i][9]<$minStations)
        {
            $min=$i;
            $minStations=$routeResult[$i][9];
        }
    }
}

if ($direct==true)
{
    $message3="Travel from ".$routeResult[$min][1]." to "
                ".$routeResult[$min][2]." on the ".$routeResult[$min][3]." line.";
}
else
{
    $message3="Travel from ".$routeResult[$min][1]." to "
                ".$routeResult[$min][4]." on the ".$routeResult[$min][3]." line.";
    if (strlen($routeResult[$min][7])<1)
    {
        $message3=$message3."<br>Travel from ".$routeResult[$min][4]." to "
                    ".$routeResult[$min][2]." on the ".$routeResult[$min][5]." line.";
    }
    else
    {
        $message3=$message3."<br>Travel from ".$routeResult[$min][4]." to "
                    ".$routeResult[$min][6]." on the ".$routeResult[$min][5]." line.";
        $message3=$message3."<br>Travel from ".$routeResult[$min][6]." to "
                    ".$routeResult[$min][2]." on the ".$routeResult[$min][7]." line.";
    }
}
echo"<p>Message 3: ".$message3;
}
echo"<form method=post action='index.php?message=".$message."'>";
echo"<p><input type=submit value='continue'>";
echo"</form>";

```

Run the web site. Again carry out tests with a variety of journeys, either on a single underground line or with one or two changes of line. In each case, check that **message3** gives a correct second journey option if alternative routes are shown in the results table.

	From station	To station	First line	First change	Second line	Second change	Third line	Map pixels	Intermediate
1	Knightsbridge	Oxford Circus	Piccadilly	Piccadilly Circus	Bakerloo			292	3
2	Knightsbridge	Oxford Circus	Piccadilly	Holborn	Central			508	7
3	Knightsbridge	Oxford Circus	Piccadilly	Green Park	Victoria			203	2
4	Knightsbridge	Oxford Circus	Piccadilly	King's Cross St Pancras	Victoria			696	10

Message 2: Travel from Knightsbridge to Green Park on the Piccadilly line.  
Travel from Green Park to Oxford Circus on the Victoria line.

Message 3: Travel from Knightsbridge to Piccadilly Circus on the Piccadilly line.  
Travel from Piccadilly Circus to Oxford Circus on the Bakerloo line.

This completes the route finding procedure in the **findRoute.php** file. We can now transfer the results back to the main web page.

Go to the section of **findRoute.php** below the results table, where the 'continue' button is displayed. Insert `/* .... */` characters to de-activate the `<form>` block as shown below. Add a **header** command which will return the program to the index page, carrying the message data within the URL.

```

    else
    {
        $message3=$message3."<br>Travel from ".$routeResult[$min][4]." to "
            ".$routeResult[$min][6]." on the ".$routeResult[$min][5]." line.";
        $message3=$message3."<br>Travel from ".$routeResult[$min][6]." to "
            ".$routeResult[$min][2]." on the ".$routeResult[$min][7]." line.";
    }
}
echo"<p>Message 3: ".$message3;
}
/*
echo"<form method=post action='index.php?message=".$message."'>";
echo"<p><input type=submit value='continue'>";
echo"</form>";
*/
header('Location: index.php?message='.$message.'&message2='
        .$message2.'&message3='.$message3);
?>
</body>
</html>

```

Save **findRoute.php** and copy it to the server.

Go now to the **index.php** file. Add lines of code near the beginning to collect the message strings from the URL when the program returns from **findRoute.php**.

```

$stationList=Stations::loadStationList();
$listCount =sizeof($stationList);
$message=$_REQUEST['message'];
$message2=$_REQUEST['message2'];
$message3=$_REQUEST['message3'];
if (!isset($message))
    $message=" ";
if (!isset($message2))
    $message2=" ";
if (!isset($message3))
    $message3=" ";
?>
<html>

```

Continuing to work in the **index.php** file, move down to the `<script>` block near the start of the `<body>` section. Add lines of code to convert the message strings to JavaScript variables.

```

var inputCount=0;
message=<? echo json_encode($message); ?>;
message2=<? echo json_encode($message2); ?>;
message3=<? echo json_encode($message3); ?>;
function setup()

```

Move down to the `setup()` function. Locate the line:  
`textarea.html(message);`  
 and replace it with the lines of code below.

```

    textarea.attribute("rows","24");
    textarea.attribute("cols","40");
    message = message.replace(/<br>/g, '\n\r');

    message2 = message2.replace(/<br>/g, '\n');
    message3 = message3.replace(/<br>/g, '\n');
    if (textWidth(message3)>10)
    {
        textarea.html(message+'\r\n\nSuggested route:\n'+message2
                       +'\r\n\nAlternative route:\n'+message3 );
    }
    else
        textarea.html(message+'\r\n\nSuggested route:\n'+message2 );
}
function draw()
{

```

Save `index.php` and copy it to the server.

Run the web site. Select a variety of journeys involving one, two or three underground lines. In each case, check that correct journey instructions are given in the information panel.



### Further development

This has been quite a substantial project covering a complex transport network. Similar applications could be developed for route finding by public transport in other major cities. Journeys might combine rail, tram and bus networks.

The project has focussed on finding shortest routes in terms of the numbers of intermediate stations between the start and destination. In many transport networks, this approach may be unsuitable. It could be better to compare journeys according to actual distance travelled, or total journey time including time spent waiting for connections.

## Summary of the object structures

### Staff

A Staff object contains the staffID which is set as an auto-number, along with the user name and password. The public method **checkPassword()** calls the private method **checkUser()** to examine each Staff object in turn, then returns an overall true/false result depending on whether the valid log-in details were found.

Staff
- staffID: integer - userName: string - password: string
+ constructor(userName, password) - checkUser(userName, password): boolean + <u>checkPassword(userName, password): boolean</u>

### Station

Objects in this class represent locations through which underground lines pass. They may be actual stations, or may be intermediate route points added to improve the accuracy of the map display. Attributes include: the x- and y-coordinates on the underground map, the name for a station, and a variable which specifies the caption display position relative to the station symbol.

Methods are provided to add a new station point, load all station objects for display on the map, edit or delete a station object. Additional methods obtain a list of station names for insertion into a drop-down selection box, and calculate the distance in screen pixels between two station points on the map.

### Line

Objects represent the London Underground lines. The attributes provide information for displaying the line on the map, including the line name, standard colour code, and whether it is displayed as a single solid line or as a double outline.

Methods are provided to add lines, and to load all line objects for use in drawing the map. Additional methods use line objects to create key displays alongside or below the map, and allow the lineID number to be obtained from the line name or *vice versa*.

### RoutePoint

Objects represent instances of particular underground lines passing through particular station points. The route points along each underground line branch are organised as a linked list, connected by pointer and backpointer values which allow the sequence to be followed in either direction. Since several underground lines may run in parallel between stations, a variable specifies the relative position where the line should be drawn on the map to avoid obscuring other lines.

Methods are provided to add or delete route points, and to load all route points so that underground lines can be displayed on the map. Methods allow the updating of pointer and backpointer values, which is necessary when changes are made to the linked lists. Methods are included for reversing the order of route points within a linked list, which may be required if additional points are added to the list.

The **stationList( )** method outputs the names of all stations on a specified underground line, to allow a search to be made for a journey route.

### Branch

Objects represent the station points along branches of underground lines. Attributes include the lineID, an identification number for the branch within that line, and arrays containing the stationID values and station names in sequence along the branch.

The **loadBranch( )** method loads station sequences for all branches of a specified underground line. Methods are provided for obtaining a station name from the stationID value, and *vice versa*.

The **countStations( )** method supervises the calculation of the number of intermediate stations between two specified station points on the same underground line. This in turn calls various methods to check for connections when the specified stations lie on different branches of the underground line. A special case is the **circleCount( )** method, which makes calculations in both directions of travel around the loop of the Circle line and chooses the journey option with least intermediate stations.

