1. According to some researches done on children below the age of 16, it was found that the height of a boy, measured in centimetres (cm), should lie within the normal range with:

minimum height = 5.3 × Age + 71

maximum height = 6.2 × Age + 87

The text file `HEIGHTDATA.TXT` contains 20 entries of the data in the following format:

<Name>, <Age>, <Height>

Task 1.1:

Write program code to:

- read the entire contents of `HEIGHTDATA.TXT`.
- determine if the boy's height lies within the normal range.
- display the contents using the format given below

Example run of the program:

Input File:

```
Ali,6,105
Bob,10,145
Charlie,15,185
```

The output generated from the input file would be:

```
Name     Age    Height    Within normal range
Ali      6      105       Yes
Bob      10     145       Yes
Charlie  15     185       No
```

Evidence 1.1:

Your program code. [6]

Evidence 1.2:

Screenshot of the output. [2]

During data entry, some of the data may have been wrongly entered with transposition errors. In the case of Charlie's, his height should have been 158 cm but was wrongly transposed and entered as 185 cm.

Task 1.2:

Write program code to:

- determine the correct height for those entries outside the normal range.
- display the amended contents using the format given below.

In cases where there are more than one possible or no possible height, print 'Re-enter data'.

Example run of the program:

Input File:

```
Ali,6,105
Bob,10,145
Charlie,15,185
Ethan,7,131
Rick,13,415
```

The output generated from the input file would be:

```
Name     Age   Height   Corrected Height
Charlie  15     185      158
Ethan     7     131      113
Rick     13     415      Re-enter data
```

Evidence 1.3:

Your program code. [6]

Evidence 1.4:

Screenshot of the output. [1]

2. The Oxford English Dictionary, published in 1989, contains 171,476 words. Instead of doing a linear search whenever we want to find a word, it is more efficient to perform a binary search.

Evidence 2.1:

Describe the binary search algorithm. [2]

Task 2.1:

Write a program `BinarySearch(lst, item)` to search for an item in the list `lst` using the binary search algorithm.

The program will:

- import the sorted list of words, given in the file `1000WORDS.TXT`, into a simple array `dataset`.
- report whether or not the item is found in the list. If found, output the index position and the list of words examined by the program during the binary search.

Evidence 2.2:

Your program code and the screenshot for the following searches:

- `BinarySearch(dataset, "WORD")`
- `BinarySearch(dataset, "WORDA")`
- `BinarySearch(dataset, "TRADE")` [8]

If we want to find all the words that start with "`TR`", we can perform a partial word search on the given dataset. The search will return a word list as follows:

```
['TRACK', 'TRADE', 'TRAIN', 'TRAVEL', 'TREE',
'TRIANGLE', 'TRIP', 'TROUBLE', 'TRUCK', 'TRUE', 'TRY']
```

Using the program written in Task 2.1 to perform a binary search for the word "`TR`"; the first word starting with "`TR`" should be "`TRADE`" found at index `889`.

We can now perform a linear search near index `889` for all the words starting with "`TR`".

Task 2.2:

Modify the code `BinarySearch(lst, item)` written in Task 2.1.

Your program will:

1. perform a partial search for the word in the list `lst` starting with the given letter(s), `item`
2. perform a linear search near the index found in step (1) to return a list of words starting with the given letter(s)

Evidence 2.3:

Your program code and the screenshot for the following searches:

- `BinarySearch(dataset, "TR")`
- `BinarySearch(dataset, "RE")` [5]

3. An `ExpressionTree` data structure is required to store 20 nodes. A linked list is maintained of all the nodes. A node contains a data value and two pointers: a left pointer and a right pointer. Items in the list are initially linked using their `LeftChild` pointers.

Each node is implemented as an instance of the class `Node`.

The class `Node` has the following properties:

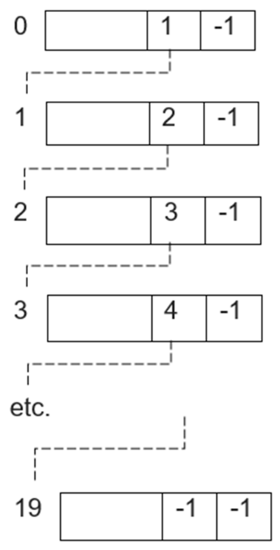
Class: `Node`		
Attributes		
Identifier	Data Type	Description
`DataValue`	`STRING`	The node data
`LeftChild`	`INTEGER`	The left node pointer
`RightChild`	`INTEGER`	The right node pointer

The `ExpressionTree` class is implemented as follows:

Class: `ExpressionTree`		
Attributes		
Identifier	Data Type	Description
`Tree`	`ARRAY[1:20] OF Node`	The tree data, initialised as a linked list
`Fringe`	`ARRAY: INTEGER`	A list to store the index of nodes traversed
`Root`	`INTEGER`	Index for the root position of the `Tree` array
`NextFreeChild`	`INTEGER`	Index for the next unused node

The index of the first available node is indicated by `NextFreeChild`. The initial value of `Root` is 0 and the initial value of `NextFreeChild` is `0`. The `Fringe` is initialised as an empty list and it will be used for node insertion to store the index for the nodes traversed.

The diagram shows the `Tree` array with the linked list to record the unused nodes.



Task 3.1

Write a program code to define the `Node` and `ExpressionTree` classes.

Evidence 3.1

Your program code for Task 3.1. [12]

The task is to store the tokens of a binary arithmetic expression in the data structure instantiated from the `ExpressionTree` class.

An arithmetic expression is a sequence of tokens that follows prescribed rules. A token may be either an operand or an operator.

A binary arithmetic operation using the standard arithmetic operators, `+ – * /`, may be in the form of operand-operator-operand. For example,

```
2 + 3
```

where `2` and `3` are operands, `+` is an operator. This expression will evaluate to a value of `5`.

Task 3.2

Write a function `IsOperator(s)` that takes in a string `s`, and returns `True` if it is a standard arithmetic operator and returns `False` if otherwise.

Evidence 3.2

Your program code for Task 3.2. [2]

An expression tree is a binary tree with the following properties:

1. Each leaf is an *operand*.
2. The root and internal nodes are *operators*.
3. Subtrees are sub-expressions, with the root being an *operator*.

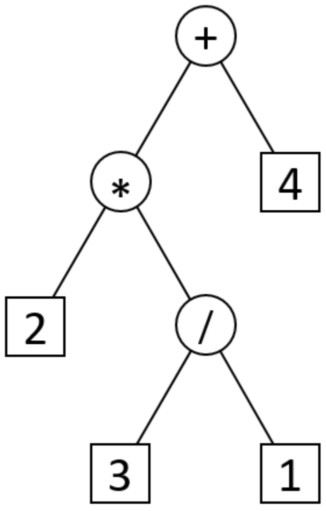
The following shows a series of commands to create and insert values into the data structure to create an expression tree.

```
CreateNewExpTree
InsertToExpTree("+")
InsertToExpTree("*")
InsertToExpTree("4")
InsertToExpTree("2")
InsertToExpTree("/")
InsertToExpTree("3")
InsertToExpTree("1")
```

The figure below shows the expression tree obtained and its **infix** expression obtained by an in-order traversal.

This expression will evaluate to `10`.

```
( 2 * ( 3 / 1 ) ) + 4
```



The following pseudocode (available in `PSEUDOCODE_TASK_3_3.TXT`) can be used to add a node to the expression tree.

```
PROCEDURE Insert(NewToken)
   IF NextFreeChild = -1 THEN    // check if tree is full
      RETURN 'Tree is Full'

   // tree is not full, safe to insert new token
   IF NextFreeChild = 0 THEN    // inserting into empty Tree
      Tree[Root].DataValue ← NewToken
      NextFreeChild ← Tree[Root].LeftChild
      Tree[Root].LeftChild ← -1

   ELSE
      // inserting into tree with existing nodes
      // starting with Root
      Current ← 0    // index of the current node
      Previous ← -1    // index of previous node
      NewNode ← Tree[NextFreeChild]    // declare new node
      NewNode.DataValue ← NewToken

   // Finding the node at which the NewNode can be inserted
      WHILE Current <> -1 THEN
         CurrNode ← Tree[Current]
         IF IsOperator(CurrNode.DataValue) THEN
         // check if CurrNode contains an operator
            IF CurrNode.LeftChild = -1 THEN
            // if LeftChild is empty, insert here
               CurrNode.LeftChild ← NextFreeChild
               NextFreeChild ← NewNode.LeftChild
               NewNode.LeftChild ← -1
               Current  ← -1

            ELIF CurrNode.RightChild = -1 THEN
               // if RightChild is empty, insert here
               CurrNode.RightChild ← NextFreeChild
               NextFreeChild ← NewNode.LeftChild
               NewNode.LeftChild ← -1
               Current  ← -1

            ELIF IsOperator(Tree[CurrNode.LeftChild].DataValue) THEN
               // if LeftChild is an operator,
               // traverse leftchild subtree
               Previous  ← Current
               Current  ← CurrNode.LeftChild
               Fringe.APPEND(Previous)
            ELIF IsOperator(Tree[CurrNode.RightChild].DataValue)THEN
               // if RightChild is an operator,
               // traverse rightchild subtree
               Previous  ← Current
               Current  ← CurrNode.RightChild
               Fringe.APPEND(Previous)

            ELSE    // traverse right subtree
               Previous  ← Fringe.POP(-1)
               Current  ← Tree[Previous].RightChild
            ENDIF

         ELSE    // no place to insert
            RETURN "Cannot be inserted"
         ENDIF
      ENDWHILE
   ENDIF
ENDPROCEDURE
```

Task 3.3

Write a code to implement the `Insert` method for the `ExpressionTree` class from this pseudocode.

You may use the text file `PSEUDOCODE_TASK_3_3.TXT` as a basis for writing your code.

Evidence 3.3

Your program code for Task 3.3. [7]

Task 3.4:

Write a code for the `Display` method for the `ExpressionTree` class which displays the contents of `Tree` in index order.

Evidence 3.4

Your program code for Task 3.4. [4]

Task 3.5

Write a sequence of program statements to:

- create an expression tree
- add the data items based on the sequence of commands given
- display the array contents

Evidence 3.5

Your program code for Task 3.5. [3]

Evidence 3.6

Screenshot showing the output from running the program in Task 3.5. [1]

Task 3.6

The infix notation can be obtained by performing an in-order traversal in the expression tree. Write a code for the `infix` method for the `ExpressionTree` class to generate the infix notation for a complete expression tree.

Evidence 3.7

Your program code for Task 3.6. [6]

Evidence 3.8

Screenshot showing the output from running the program in Task 3.6. [1]

Task 3.7

Write a code for the `calculate` method to evaluate and return the numerical answer for the expression, rounded off to 2 decimal places.

Evidence 3.9

Your program code for Task 3.7. [3]

Evidence 3.10

Screenshot showing the output from running the program in Task 3.7. [1]

4. Minesweeper is a type of single-player puzzle game in which the player continuously selects a cell in a square grid. Each cell contains either a bomb or a value showing the number of bombs in the neighbouring cell. (Neighbouring cells are those adjacent horizontally, vertically or diagonally.)

If the player selects a cell that is a bomb, it 'explodes' and he loses the game. The number of cells the player has selected without exploding a bomb will be the player's score.

You are required to write a program code to generate a minesweeper grid, randomly position the bombs and populate all the other cells with the values indicating the number of bombs in the neighbouring cells. Without revealing the minesweeper grid to the player, the program should prompt the player to select the cells one by one. His score will be the number of cells opened before hitting a bomb.

```
X  2  1  1
1  3  X  2
0  2  X  3
0  1  2  X
```

Task 4.1:

Write a program code to generate and display an empty square grid of size *n*, ie *n* rows by *n* columns. The minesweeper grid for *n* = 5 is as shown below:

```
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
0  0  0  0  0
```

Your code should use a suitable data structure and fixed loop(s) to display the grid.

Evidence 4.1:

Your program code and screenshot of an empty grid of size 5. [3]

Task 4.2:

Write a program code to randomly place a bomb, represented by "X", within the grid. Populate all the neighbouring cells by increasing their values to 1 to indicate the presence of this one bomb in the neighbourhood.

Evidence 4.2:

Your program code and two different screenshots of the grid (*n*=5). [5]

Task 4.3:

Modify the code written in Task 4.2 to randomly place two bombs within the grid. Populate all the neighbouring cells with the correct values to indicate the presence of the bombs in the neighbourhood.

Evidence 4.3:

Your program code and the screenshot of the grid (*n*=5) with 2 bombs. [4]

Task 4.4:

Modify the code written in Task 4.3 to generate *k* numbers of bombs within a grid of size *n* and correctly display all the values in the neighbouring cells surrounding the bombs.

Evidence 4.4:

Your program code and the screenshots of the minesweeper grids for the following levels of difficulty.

- Beginner (grid size *n*=5; no. of bombs *k*=3)
- Intermediate (grid size *n*=6; no. of bombs *k*=8)
- Expert (grid size *n*=8; no. of bombs *k*=20) [8]

Task 4.5:

Write a program code to play the minesweeper game. Your code will:

- prompt the player to select the level of difficulty
- generate the Minesweeper grid
- display a "blank" grid with '–' for each of the cell
- prompt the player to input the coordinates of a cell he wishes to open
    - If the opened cell is a bomb ("X"), declare "Game Over!", show the grid and display the player's score.
    - If the opened cell is not a bomb, show the updated grid with the opened cell, increase the player's score by 1 and continue with the game.
- declare "You have Won!" when the player has opened all the possible cells and display the player's score.

[Sample screenshot of a typical game]:

```
Enter your cell you want to open :
X (1 to 5) : 2
Y (1 to 5) : 3

- - - - -

- - 1 - -

0 - - - -

- - 1 - -

- - - - -

Your score is :  3
```

```
Enter your cell you want to open :
X (1 to 5) : 2
Y (1 to 5) : 4

X 1 1 1 1

1 1 1 X 1

0 0 1 1 1

0 1 1 1 0

0 1 X 1 0

Game Over! You've hit the bomb at : (2,4).
Your score is :  3
Do you want to try again?
```

Evidence 4.5:

Your program code and a screenshot of a game. [10]