In [16]:

```
import numpy as np
import graphviz

class Activation:
    SIGMOID = "sigmoid"
    LINEAR = "linear"
    RELU = "relu"
    SOFTMAX = "softmax"

class Layer:
    def __init__(self, n_input, n_nodes):
        self.weights = []
        self.n_input = n_input
        self.n_nodes = n_nodes
        self.activations = ""
        self.input = []
        self.output = []


class FFNN:
    def __init__(self, file_name):  # read model from file
        with open(file_name, "r") as f:
            # define attributes
            self.filename = file_name
            self.layers = []
            self.bias = 1
            self.input = []
            self.output = []
            self.sigma = []
            self.learning_rate = 0.001

            # read n_input
            self.n_input = int(f.readline()[0])

            # read node per layer
            line = f.readline().strip(" \n").split(" ")
            self.n_layers = len(line) - 1
            for i in range(1, self.n_layers + 1):
                self.layers.append(Layer(int(line[i-1]), int(line[i])))

            # read input
            f.readline()
            for i in range(self.n_input):
                input = []
                line = f.readline().strip(" \n").split(" ")
                input.append(1)  # bias
                for j in range(len(line)):
                    input.append(float(line[j]))
                self.input.append(input)

            # read weight for every layer
            for i in range(self.n_layers):
                f.readline()
                for j in range(self.layers[i].n_input + 2):
                    weight = []
                    line = f.readline().strip(" \n").split(" ")
                    for k in range(len(line)):
                        if (j == 0):
                            self.layers[i].activations = str(line[k])
```

```
                    else:
                        weight.append(float(line[k]))
                if j!=0:
                    self.layers[i].weights.append(weight)

    def showLayer(self):
        for i in range(self.n_layers):
            print("layer", i, ":", self.layers[i].weights)
        return self.layers

    def forward_propagation(self):
        # The first input layer
        self.layers[0].input = self.input
        print(f"----- Layer: 0 -----")
        for idx, val in enumerate(self.input):
            print(f"Input {idx}: {val[1:]}")

        # All layers
        for i in range(self.n_layers):
            print(f"------- Layer: {i+1} -------")
            print(f"Activation : {self.layers[i].activations}")

            # case first input layer
            if (i != 0):
                for j in range(self.n_input):
                    bias_input = [1]
                    bias_input.extend(self.layers[i-1].output[j])
                    self.layers[i].input.append(bias_input)

            # calculate sigma
            self.layers[i].output  = np.dot(
                self.layers[i].input, self.layers[i].weights)

            # activation function
            for j in range(len(self.layers[i].output)):
                print(f"Input-{j} ==>")
                for k in range(len(self.layers[i].output[j])):

                    x = self.layers[i].output[j][k]
                    result = 0
                    if (self.layers[i].activations == Activation.LINEAR):
                        result = format(self.linear(x), ".2f")
                    elif (self.layers[i].activations == Activation.SIGMOID):
                        result = format(self.sigmoid(x), ".2f")
                    elif (self.layers[i].activations == Activation.RELU):
                        result = format(self.relu(x), ".2f")
                    else: #SOFTMAX
                        result = format(self.softmax(x), ".2f")

                    self.layers[i].output[j][k] = result
                    self.sigma.append(x)

                    print(f"Node {k+1}")
                    print(f"Sigma   : {x}")
                    print(f"Result  : {self.layers[i].output[j][k]}")

        # current output
        self.output = self.layers[self.n_layers-1].output.copy()

    def sigmoid(self, x):
        return 1/(1+np.exp(-x))
```

```
    def relu(self, x):
        return np.maximum(x, 0)

    def linear(self, x):
        return x

    def softmax(self, x):
        net_h = np.array(x)
        numerator = np.exp(net_h)
        denominator = np.sum(np.exp(x))
        softmax_output = numerator / denominator
        return softmax_output

    def show_output(self):
        print("Prediction")
        print(self.output)

    def show_layer(self):
        for x in range(self.n_layers):
            print(f"Layer: {self.layers[x].output}")

    def draw_model(self):
        f = graphviz.Digraph('Feed Forward Neural Network', filename=self.filename)
        f.attr('node', shape='circle', width='1.0')
        f.edge_attr.update(arrowhead='vee', arrowsize='2')

        for i in range(self.n_layers):
            if i == 0:
                for j in range(len(self.layers[i].weights)): #count weights
                    for k in range(len(self.layers[i].weights[j])): #output node
                        if j==0:
                            f.edge(f"bx{j}", f"h{i+1}_{k}",
                                   f"{self.layers[i].weights[j][k]}")
                        else:
                            f.edge(f"x{j}", f"h{i+1}_{k}",
                                   f"{self.layers[i].weights[j][k]}")
            else:
                for j in range(len(self.layers[i].weights)): #count weights
                    for k in range(len(self.layers[i].weights[j])): #output node
                        if j==0:
                            f.edge(f"bh{i}", f"h{i+1}_{k}",
                                   f"{self.layers[i].weights[j][k]}")
                        else:
                            f.edge(f"h{i}_{j-1}", f"h{i+1}_{k}",
                                   f"{self.layers[i].weights[j][k]}")

        print(f.source)
        f.render(directory='model').replace('\\', '/')
```

In [17]:

```
ffnn = FFNN("model1.txt")
ffnn.forward_propagation()
ffnn.show_output()
ffnn.draw_model()
```

```
----- Layer: 0 -----
Input 0: [0.0, 0.0]
------- Layer: 1 -------
Activation : sigmoid
Input-0 ==>
Node 1
Sigma   : -10.0
Result  : 0.0
Node 2
Sigma   : 30.0
Result  : 1.0
------- Layer: 2 -------
Activation : sigmoid
Input-0 ==>
Node 1
Sigma   : -10.0
Result  : 0.0
Prediction
[[0.]]
digraph "Feed Forward Neural Network" {
        edge [arrowhead=vee arrowsize=2]
        node [shape=circle width=1.0]
        bx0 -> h1_0 [label=-10.0]
        bx0 -> h1_1 [label=30.0]
        x1 -> h1_0 [label=20.0]
        x1 -> h1_1 [label=-20.0]
        x2 -> h1_0 [label=20.0]
        x2 -> h1_1 [label=-20.0]
        bh1 -> h2_0 [label=-30.0]
        h1_0 -> h2_0 [label=20.0]
        h1_1 -> h2_0 [label=20.0]
}
```

In [18]:

```
ffnn = FFNN("model2.txt")
ffnn.forward_propagation()
ffnn.show_output()
ffnn.draw_model()
```

```
----- Layer: 0 -----
Input 0: [0.0, 1.0]
------- Layer: 1 -------
Activation : relu
Input-0 ==>
Node 1
Sigma   : 1.0
Result  : 1.0
Node 2
Sigma   : 0.0
Result  : 0.0
------- Layer: 2 -------
Activation : relu
Input-0 ==>
Node 1
Sigma   : 1.0
Result  : 1.0
Prediction
[[1.]]
digraph "Feed Forward Neural Network" {
        edge [arrowhead=vee arrowsize=2]
        node [shape=circle width=1.0]
        bx0 -> h1_0 [label=0.0]
        bx0 -> h1_1 [label=-1.0]
        x1 -> h1_0 [label=1.0]
        x1 -> h1_1 [label=1.0]
        x2 -> h1_0 [label=1.0]
        x2 -> h1_1 [label=1.0]
        bh1 -> h2_0 [label=0.0]
        h1_0 -> h2_0 [label=1.0]
        h1_1 -> h2_0 [label=-2.0]
}
```

In [ ]:

```
```

In [ ]:

```
```