Task 1

A buffer is a portion of the computer's memory used to store data temporarily while it is being transferred to an external device. As the buffer manages data in a First In First Out (FIFO) manner, it would be appropriate to operate the buffer using a queue data structure.

The task is to design the queue data structure to be used for the buffer operations.

Task 1.1

Write program code to implement a queue data structure with the following specifications:

- `make_queue()` constructs an empty queue
- `enqueue(queue,item)` appends an item to the queue
- `dequeue(queue)` returns and removes the next item in the queue
- `size(queue)` returns the size of the queue.

All three of `enqueue(queue,item)`, `dequeue(queue)` and `size(queue)` will return `None` when the queue is empty.

Task 1.2

Write program code to:

- construct a new empty queue `buffer`
- add three pieces of data into `buffer`
- remove and output two elements from `buffer`
- output the size of `buffer`.

Task 2

Infix notation is a form of notation for arithmetic, logic, and algebraic expressions where operators are written in-between the operands they act upon. It's the way expressions are most commonly written and understood by people such as the expression

```
5 × (6 + 2) − 12 ÷ 4
```

Postfix notation, also known as Reverse Polish notation is a form of notation where the operator comes after the operands. The postfix notation does not require parentheses for the order of operation and is useful in certain computational contexts. For example, the aforementioned expression can be written in postfix notation as

```
5 6 2 + * 12 4 / -
```

When reading an expression in postfix notation, the values are scanned from left to right until an operator is found. The two values preceding the operator are then evaluated based on the operation represented by the operator. This continues until all operations are complete. For example, the aforementioned expression in postfix form can be evaluated in the following order:

```
5 6 2 + * 12 4 / -
  5 8 * 12 4 / -
    40 12 4 / -
       40 3 -
```

A stack can be used to evaluate an arithmetic expression in infix notation by first converting it to postfix notation, before evaluating the expression in postfix notation to obtain the final result.

The task is to evaluate mathematical expressions using a stack.

Task 2.1

Implement a `Stack` class with the following properties and methods:

- `data` – an array of size 20 that holds stack elements of the string data type, with array index initialised to 1
- `top` – index of the `data` array for the element at the top of the stack, initialised to an appropriate value for each new instance of `Stack`.
- `is_empty` – returns an appropriate Boolean value to indicate whether there are elements stored on the stack
- `is_full` – returns an appropriate Boolean value to indicate whether the stack is full
- `push` – inserts an element onto the stack
- `pop` – removes and returns the last inserted element from the stack
- `peek` – returns a copy of the last inserted element from the stack without removing it
- `size` – returns the number of elements stored on the stack
- `display` – outputs the contents of the stack with the top of the stack clearly indicated.

Task 2.2

Write program code for the function

```
FUNCTION Infix2Postfix (infixExpression: STRING) RETURNS STRING
```

that will implement an algorithm to convert an expression in infix notation to postfix notation.

The algorithm is as follows:

Step 1
Create an empty stack called `opStack` for keeping operators.

Step 2
Scan the expression in infix notation from left to right.

- If an operand is detected, append it to the end of the output list.
- If a left parenthesis is detected, push it onto `opStack`.
- If a right parenthesis is detected, pop the top element of `opStack` until the corresponding left parenthesis is removed. Append each operator to the end of the output list.
- If an operator is detected push it onto `opStack`. However, first remove any operators already on the `opStack` that have higher or equal mathematical evaluation precedence and append them to the output list

Step 3
When the input expression has been completely processed, check the `opStack`. Any operators still on the `opStack` can be removed and appended to the end of the output list.

Task 2.3

Write program code for the function

```
FUNCTION PostfixEval (postfixExpression: STRING) RETURNS FLOAT
```

that will implement an algorithm for evaluating expressions in the postfix notation

The algorithm is as follows:

Step 1
Create an empty stack called `operandStack`.

Step 2
Scan the expression in postfix notation from left to right.

- If an operand is detected, convert it from a string to an integer and push the value onto the `operandStack`.
- If an operator is detected, it will need two operands. Pop the tot two elements from the `operandStack`. The first popped element is the second operand and the second popped

is the first operand. Perform the arithmetic operation. Push the result back onto the `operandStack`.

Step 3

When the input expression has been completely processed, the result is on the `operandStack`. Pop the result from the `operandStack` and return the value.

Task 2.4

Write program code to read the infix expressions from the file `INFIX.txt` and output the corresponding postfix expressions and evaluated results.