Stacks

Stacks are a subclass of Linear Lists; all access to a stack is restricted to one end of the list, called the *top of stack*. Visually, picture a stack of books, coins, plates, etc. Addition of books, plates, etc. occurs at the top of the stack; removal also occurs at the top of the stack.

A Stack is an ordered (by position, not by value) collection of data (usually homogeneous), with the following operations defined on it:

Operation	**Description**
Initialize	Initialize internal structure; create empty stack
Push	Add new element to top of stack
Pop	Remove top element from stack
Empty	True iff stack has no elements
StackTop	Returns copy of top element of stack (without popping it)
Size	Returns number of elements in the stack

An array-based stack requires we know two values *a priori*: the type of data contained in the stack, and the size of the array. For our implementation, we will assume that the stack stores integer numbers and can store 10 numbers.

The stack itself is a structure containing two elements: **Data**, the array of data, and **Top**, an index that keeps track of the current *top of stack*; that is, it tells us where data is added for a Push and removed for a Pop.

Initialize

Preconditions : none
Post-Conditions: Stack, Top set to 1

Pseudo-Code :

```
Set Top to 1
Return Stack_Array and Top to the user.
```

Push

Preconditions : Non-Full Stack, Element to Push
Post-Conditions: Stack with element pushed onto it

Pseudo-Code :

```
If Stack is Full Then
        Output "Overflow, Stack is full, cannot push."
Else
```

```
        Place Element in Stack(Top)
        Increment Top
        Return Stack and Top to the user.
```

Pop

Preconditions : Non-Empty Stack
Post-Conditions: Stack with top element popped off, element popped off

Pseudo-Code :

```
    If Stack is Empty Then
        Output "Underflow, Stack is empty, cannot pop."
    Else
        Top:= Top-1;
        Return element in Stack(Top)
```

Empty

Preconditions : Stack
Post-Conditions: Determines if the stack is empty

Pseudo-Code :

```
    If Top = 1 Then
        Return Empty_Stack := True
    Else
        Return Empty_Stack := False
```

StackTop

Preconditions : Stack
Post-Conditions: Return the top element in a non-empty stack

Pseudo-Code :

```
    If Top = 1 Then
        Output "Stack is Empty – Cannot get Top"
    Else
        Return Stack(Top-1)
```

Size

Preconditions : Stack
Post-Conditions: Return the size of the stack

Pseudo-Code :

Return (Top-1)

Infix to Postfix Conversion

Preconditions: A non-empty input string containing the expression in infix form
Postconditions: A string in postfix form that is equivalent to the infix expression

Pseudocode:
1. Create a user Stack
2. Get the infix expression from the user as a string, say Infix
3. Check if the paranthesis are balanced as follows:
    For I in 1.. length(Infix) do
        i. If Infix(I) = '(' then Push onto the Stack
        ii. If Infix(I) = ')' then Pop one element form the Stack
4. If Stack is non-empty
    a. Display "non-balanced expression"
    b. Goto 2
5. Create a new string Postfix
6. Set Postfix_Index to 1
7. For I in 1 .. Length(Infix)
    a. If Infix(I) is an operand, append it to postfix string as follows:
        i. Postfix(Postfix_Index) := Infix(I);
        ii. Postfix_Index:=Postfix_Index + 1;
    b. If the Infix(I) is an operator, process operator as follows
        1. Set done to false
        2. Repeat
            a. If Stack is empty or Infix(I) is '(' then
                i. push Infix(I) onto stack
                ii. set done to true
            b. Else if precedence(Infix(I)) > precedence(top operator)
                i. Push Infix(I) onto the stack (ensures higher precedence operators evaluated first)
                ii. set done to true
            c. Else
                i. Pop the operator stack
                ii. If operator popped is '(', set done to true
                iii. Else append operator popped to postfix string
        3. Until done
8. While Stack is not empty
    a. Pop operator
    b. Append it to the postfix string
9. Return Postfix

Queues

Queues are a subclass of Linear Lists, which maintain the First-In-First-Out order of elements. Insertion of elements is carried out at the 'Tail' of the queue and deletion is carried out at the 'Head' of the queue.

A queue is an ordered (by position, not by value) collection of data (usually homogeneous), with the following operations defined on it:

Operation Description

Initialize	Initialize internal structure; create an empty queue
Enqueue	Add new element to the tail of the queue
Dequeue	Remove an element from the head of the queue
Empty	True iff the queue has no elements
Full	True iff no elements can be inserted into the queue
Size	Returns number of elements in the queue
Display	Display the contents of the Queue

An array-based queue requires us to know two values *a priori*: the type of data contained in the queue, and the size of the array. For our implementation, we will assume that the queue stores integer numbers and can store 10 numbers.

The queue itself is a structure containing three elements: **Data**, the array of data, **Head**, an index that keeps track of the first element in the queue (location where data is removed from the queue), and **Tail**, an index that keeps track of the last element in the queue (location where elements are inserted into the queue).

Initialize

Preconditions : none
Post-Conditions: Queue, Head, Tail set to 1

Pseudo-Code :

    Set Head to 1
    Set Tail to 1
    Return the Queue to the user.

Enqueue

Preconditions : Non-Full Queue, Element to insert
Post-Conditions: Queue with the element appended to it

Pseudo-Code :

    If Queue is Full (Tail = Size of Queue + 1) Then

Output “Overflow, Queue is full, cannot Enqueue.”
Else
Place Element in Queue(Tail)
Increment Tail (Tail = Tail + 1)
Return the queue to the user.

Dequeue

Preconditions : Non-Empty Queue
Post-Conditions: Queue with element at Head removed, element that is dequeued

Pseudo-Code :

```
If Queue is Empty (Head = Tail) Then
    Output “Underflow, Queue is empty, cannot dequeue.”
Else
    Element := Queue(Head);
    Move all the elements from head+1 to Size of Queue one step to the left
    Return Element
```

Empty

Preconditions : Queue
Post-Conditions: Determines if the queue is empty

Pseudo-Code :

```
If Head = Tail Then
    Return Empty_Queue := True
Else
    Return Empty_Queue:= False
```

Full

Preconditions : Queue
Post-Conditions: Return True if the Queue is full

Pseudo-Code :

```
If Tail = Queue_Size+1 Then
    Return True
Else
    Return False
```

Size

Preconditions : Queue
Post-Conditions: Return the number of elements in the queue

Pseudo-Code :

```
Return (Tail - Head)
```

Display

Preconditions : Queue
Post-Conditions: Display the contents of the queue

Pseudo-Code :

```
If head < 1 then
    Lb :=1;
Else
    Lb := Head;
    If tail > max_queue_ size + 1 then
        Ub := max_queue_size;
    Else
        Ub := Tail;
        For I:= Lb to Ub
            Display Queue(I)
```

Jayakanth Srinivasan
I. Kristina Lundqvist