Chapter 19: Computational thinking and problem solving

19.1 Algorithms

Keyterms:

- Binary search
  a method of searching an ordered list by testing the value of the middle item in the list and rejecting the half of the list that does not contain the required value.

- Insertion sort
  a method of sorting data in an array into alphabetical or numerical order by placing each item in turn in the correct position in the sorted list.

- Binary tree
  a hierarchical data structure in which each parent node can have a maximum of two child nodes.

- Graph
  a non-linear data structure consisting of nodes and edges.

- Dictionary
  an abstract data type that consists of pairs, a key and a value, in which the key is used to find the value.

- Big O notation
  a mathematical notation used to describe the performance or complexity of an algorithm.

19.1.1 Understanding Linear and Binary searching methods

Linear Search

- This method works for a list in which the items can be **stored in any order**.
- But as the **size** of the list **increases**, the **average time** taken to retrieve an item **increases** correspondingly.

Identifier	Description
myList	Array to be searched
upperBound	Upper bound of the array
lowerBound	Lower bound of the array
index	Pointer to current array element
item	Item to be found
found	Flag to show when item has been found

```
DECLARE myList : ARRAY[0:9] OF INTEGER
DECLARE upperBound : INTEGER
DECLARE lowerBound : INTEGER
DECLARE index : INTEGER
DECLARE item : INTEGER
DECLARE found : BOOLEAN
upperBound ← 9
lowerBound ← 0
OUTPUT "Please enter item to be found"
INPUT item
found ← FALSE
index ← lowerBound
REPEAT
    IF item = myList[index]
      THEN
        found ← TRUE
    ENDIF
    index ← index + 1
UNTIL (found = TRUE) OR (index > upperBound)
IF found
  THEN
    OUTPUT "Item found"
  ELSE
    OUTPUT "Item not found"
ENDIF
```

```
'VB program for Linear Search
 Module Module1
    Public Sub Main()
        Dim index As Integer
        Dim item As Integer
        Dim found As Boolean
        'Create array to store all the numbers
        Dim myList() As Integer = New Integer() {4, 2, 8, 17, 9, 3, 7, 12, 34, 21}
        'enter item to search for
        Console.Write("Please enter item to be found ")
        item = Integer.Parse(Console.ReadLine())
        For index = 0 To myList.Length - 1
            If (item = myList(index)) Then
                found = True
            End If
        Next
        If (found) Then
            Console.WriteLine("Item found")
        Else : Console.WriteLine("Item not found")
        End If
        Console.ReadKey()
    End Sub
End Module
```


Binary Search

- A binary search is more efficient if a list is **already sorted**.
- The value of the **middle item** in the list is **first tested** to see if it matches the required item, and the half of the list that does not contain the required item is discarded.
- Then, the next item of the list to be tested is the middle item of the half of the list that was kept.
- This is repeated until the required item is found or there is nothing left to test.

For example, consider a list of the letters of the alphabet.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

To find the letter **W** using a **linear search** there would be **23 comparisons**.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=	=			
W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W	W			
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23			

To find the letter **W** using a **binary search** there could be just **3 comparisons**.

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
												=							=			=			
												W							W			W			
												1							2			3			

Identifier	Description
myList	Array to be searched
upperBound	Upper bound of the array
lowerBound	Lower bound of the array
index	Pointer to current array element
item	Item to be found
found	Flag to show when item has been found

```
DECLARE myList : ARRAY[0:9] OF INTEGER
DECLARE upperBound : INTEGER
DECLARE lowerBound : INTEGER
DECLARE index : INTEGER
DECLARE item : INTEGER
DECLARE found : BOOLEAN
upperBound ← 9
lowerBound ← 0
OUTPUT "Please enter item to be found"
INPUT item
found ← FALSE
REPEAT
    index ← INT ( (upperBound + lowerBound) / 2 )
    IF item = myList[index]
      THEN
        found ← TRUE
    ENDIF
    IF item > myList[index]
      THEN
        lowerBound ← index + 1
    ENDIF
    IF item < myList[index]
      THEN
        upperBound ← index - 1
    ENDIF
UNTIL (found = TRUE) OR (lowerBound = upperBound)
IF found
  THEN
    OUTPUT "Item found"
  ELSE
    OUTPUT "Item not found"
ENDIF
```

```
index = (upperBound + lowerBound)\2
  ▶ If (item = myList(index)) Then
      found = True
    End If
  ▶ If item > myList(index) Then
      lowerBound = index + 1
    End if
  ▶ If item < myList(index) Then
      upperBound = index -1
    End if
```


ACTIVITY 19C

In your chosen programming language, write a short program to complete the binary search.
Use this sample data:
16, 19, 21, 27, 36, 42, 55, 67, 76, 89
Search for the values 19 and 77 to test your program.

```
'VB program for Binary Search
Module Module1
        Public Sub Main()
                Dim index, lowerBound, upperBound As Integer
                Dim item As Integer
                Dim found As Boolean

                'create array to store all the numbers
                Dim myList() As Integer = New Integer() {16, 19, 21, 27, 36, 42, 55, 67, 76, 89}

                'enter item to search for
                Console.Write("Please enter item to be found ")
                item = Integer.Parse(Console.ReadLine())

                found = False
                lowerBound = 0
                upperBound = myList.Length - 1

                Do
                        index = (upperBound + lowerBound) / 2

                        If (item = myList(index)) Then
                                found = True
                        End If

                        If item > myList(index) Then
                                lowerBound = index + 1
                        End If

                        If item < myList(index) Then
                                upperBound = index - 1
                        End If

                Loop Until (found) Or (lowerBound > upperBound)

                If (found) Then
                        Console.WriteLine("Item found")
                Else : Console.WriteLine("Item not found")
                End If

                Console.ReadKey()
        End Sub
End Module
```

19.1.2 Understanding Insertion and Bubble sorting methods

Bubble Sort

- Sorts data in an array into **alphabetical** or **numerical** order by **comparing adjacent items**
  and **swapping** them if they are in the **wrong order**.
- The bubble sort works well for **short lists** and **partially sorted lists**.

Identifier	Description
myList	Array to be searched
upperBound	Upper bound of the array
lowerBound	Lower bound of the array
index	Pointer to current array element
swap	Flag to show when swaps have been made
top	Index of last element to compare
temp	Temporary storage location during swap

```
DECLARE myList : ARRAY[0:8] OF INTEGER
DECLARE upperBound : INTEGER
DECLARE lowerBound : INTEGER
DECLARE index : INTEGER
DECLARE swap : BOOLEAN
DECLARE temp : INTEGER
DECLARE top : INTEGER
upperBound ← 8
lowerBound ← 0
top ← upperBound
REPEAT
   FOR index = lowerBound TO top - 1
        Swap ← FALSE
        IF myList[index] > myList[index + 1]
          THEN
          temp ← myList[index]
          myList[index] ← myList[index + 1]
          myList[index + 1] ← temp
          swap ← TRUE
        ENDIF
    NEXT
    top ← top -1
UNTIL (NOT swap) OR (top = 0)
```

```
'VB program for bubble sort
 Module Module1
    Sub Main()
        Dim myList() As Integer = New Integer() {70, 46, 43, 27, 57, 41, 45, 21, 14}
        Dim index, top, temp As Integer
        Dim swap As Boolean
        top = myList.Length - 1
        Do
            swap = False
            For index = 0 To top - 1 Step 1
                If myList(index) > myList(index + 1) Then
                    temp = myList(index)
                    myList(index) = myList(index + 1)
                    myList(index + 1) = temp
                    swap = True
                End If
            Next
            top = top - 1
        Loop Until (Not swap) Or (top = 0)    ' Post-condition loop
        'output the sorted array
        For index = 0 To myList.Length - 1
            Console.Write(myList(index) & " ")
        Next
        Console.ReadKey() 'wait for keypress
    End Sub
End Module
```

First pass of bubble sort

All nine elements compared and five swaps:

	index	myList								
	[0]	27	19	19	19	19	19	19	19	19
	[1]	19	27	27	27	27	27	27	27	27
	[2]	36	36	36	36	36	36	36	36	36
	[3]	42	42	42	42	16	16	16	16	16
	[4]	16	16	16	16	42	42	42	42	42
	[5]	89	89	89	89	89	89	21	21	21
	[6]	21	21	21	21	21	21	89	16	16
	[7]	16	16	16	16	16	16	16	89	55
top →	[8]	55	55	55	55	55	55	55	55	89

Second pass of bubble sort

Eight elements compared and three swaps:

	index	myList							
	[0]	19	19	19	19	19	19	19	19
	[1]	27	27	27	27	27	27	27	27
	[2]	36	36	36	16	16	16	16	16
	[3]	16	16	16	36	36	36	36	36
	[4]	42	42	42	42	42	21	21	21
	[5]	21	21	21	21	21	42	16	16
	[6]	16	16	16	16	16	16	42	42
top →	[7]	55	55	55	55	55	55	55	55
	[8]	89	89	89	89	89	89	89	89

Third pass of bubble sort

Seven elements compared and three swaps:

	index	myList						
	[0]	19	19	19	19	19	19	19
	[1]	27	27	27	27	27	27	27
	[2]	16	36	36	16	16	16	16
	[3]	36	16	16	36	36	36	36
	[4]	21	42	42	42	42	21	21
	[5]	16	21	21	21	21	42	16
top →	[6]	42	16	16	16	16	16	42
	[7]	55	55	55	55	55	55	55
	[8]	89	89	89	89	89	89	89

Fourth pass of bubble sort

Six elements compared and three swaps:

	index	myList					
	[0]	19	19	19	19	19	19
	[1]	27	27	16	16	16	16
	[2]	16	16	27	27	27	27
	[3]	36	36	36	36	21	21
	[4]	21	21	21	21	36	16
top →	[5]	16	16	16	16	16	36
	[6]	42	42	42	42	42	42
	[7]	55	55	55	55	55	55
	[8]	89	89	89	89	89	89

Fifth pass of bubble sort

Five elements compared and three swaps:

	index	myList				
	[0]	19	16	16	16	16
	[1]	16	19	19	19	19
	[2]	27	27	21	21	21
	[3]	21	21	27	27	16
top →	[4]	16	16	16	16	27
	[5]	36	36	36	36	36
	[6]	42	42	42	42	42
	[7]	55	55	55	55	55
	[8]	89	89	89	89	89

Sixth pass of bubble sort

Four elements compared and one swap:

	index	myList			
	[0]	16	16	16	16
	[1]	19	19	19	19
	[2]	21	21	21	16
top →	[3]	16	16	16	21
	[4]	27	27	27	27
	[5]	36	36	36	36
	[6]	42	42	42	42
	[7]	55	55	55	55
	[8]	89	89	89	89

Seventh pass of bubble sort

Three elements compared and one swap:

	index	myList		
	[0]	16	16	16
	[1]	19	19	16
top →	[2]	16	16	19
	[3]	21	21	21
	[4]	27	27	27
	[5]	36	36	36
	[6]	42	42	42
	[7]	55	55	55
	[8]	89	89	89

Eighth pass of bubble sort

Two elements compared and no swaps:

	index	myList	
	[0]	16	16
top →	[1]	16	16
	[2]	19	19
	[3]	21	21
	[4]	27	27
	[5]	36	36
	[6]	42	42
	[7]	55	55
	[8]	89	89

Insertion Sort

- An insertion sort will also work well for **short lists** and **partially sorted lists**.
- It sorts data in a list into **alphabetical** or **numerical** order by placing each item in turn in the **correct position** in a **sorted list**.
- It works well for **incremental sorting**, where **elements** are **added** to a **list one at a time** over an extended period **while keeping the list sorted**.

Identifier	Description
myList	Array to be searched
upperBound	Upper bound of the array
lowerBound	Lower bound of the array
index	Pointer to current array element
key	Element being placed
place	Position in array of element being moved

```
DECLARE myList : ARRAY[0:8] OF INTEGER
DECLARE upperBound : INTEGER
DECLARE lowerBound : INTEGER
DECLARE index : INTEGER
DECLARE key : BOOLEAN
DECLARE place : INTEGER
upperBound ← 8
lowerBound ← 0
FOR index ← lowerBound + 1 TO upperBound
    key ← myList[index]
    place ← index - 1
    IF myList[place] > key
      THEN
        WHILE place >= lowerBound AND myList[place] > key
            temp ← myList[place + 1]
            myList[place + 1] ← myList[place]
            myList[place] ← temp
            place ← place - 1
        ENDWHILE
        myList[place + 1] ← key
    ENDIF
NEXT index
```

```
For index = lowerBound + 1 To upperBound
    myKey = myList(index)
    place = index - 1
    If myList(place) > myKey Then
        While (place >= lowerBound) And (myList(place) > myKey)
            temp = myList(place + 1)
            myList(place + 1) = myList(place)
            myList(place) = temp
            place = place - 1
        End While
        myList(place + 1) = myKey
    End If
Next
```

	Index of element being checked												
myList	1		2	3	4		5	6		7		8	
[0]	27	19	19	19	19	16	16	16	16	16	16	16	16
[1]	19	27	27	27	27	19	19	19	19	19	16	16	16
[2]	36	36	36	36	36	27	27	27	21	21	19	19	19
[3]	42	42	42	42	42	36	36	36	27	27	21	21	21
[4]	16	16	16	16	16	42	42	42	36	36	27	27	27
[5]	89	89	89	89	89	89	89	89	42	42	36	36	36
[6]	21	21	21	21	21	21	21	21	89	89	42	42	42
[7]	16	16	16	16	16	16	16	16	16	16	89	89	55
[8]	55	55	55	55	55	55	55	55	55	55	55	55	89

- The element shaded **blue** is being **checked** and **placed** in the **correct position**.
- The elements shaded **yellow** are the **other elements** that also **need to be moved** if the element being checked is **out of position**.
- When **sorting** the same array, myList, the **insert sort** made **21 swaps** and the **bubble sort** shown in Chapter 10 made **38 swaps**.
- The **insertion sort** performs **better** on **partially sorted lists** because, when each element is **found** to be in the **wrong order** in the list, it is **moved** to approximately the **right place** in the list.
- The **bubble sort** will **only swap** the element in the **wrong order** with its **neighbour**.
- As the **number** of **elements** in a list **increases**, the time taken to sort the list **increases**.

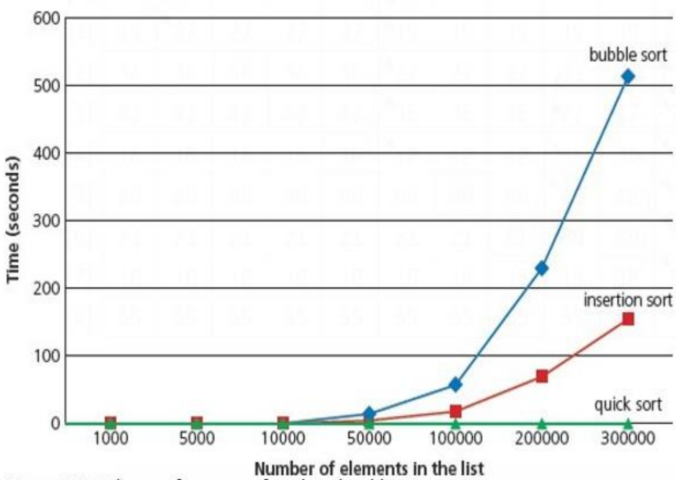


Figure 19.4 Time performance of sorting algorithms

- It has been shown that, as the **number of elements increases**, the performance of the **bubble sort deteriorates faster** than the insertion sort.

ACTIVITY 19D

- In your chosen programming language write a short program to complete the insertion sort.
- Use this sample data: 4, 46, 43, 27, 57, 41, 45, 21, 14

```
'VB program for insertion sort
Module Module1
    Sub Main()
        Dim myList() As Integer = New Integer() {4, 46, 43, 27, 57, 41, 45, 21, 14}
        Dim index, lowerBound, upperBound, place, myKey, temp As Integer
        upperBound = myList.Length - 1
        lowerBound = 0

        For index = lowerBound + 1 To upperBound
            myKey = myList(index)
            place = index – 1

            If myList(place) > myKey Then
                Do While (place >= lowerBound) And myList(place) >
myKey
                    temp = myList(place + 1)
                    myList(place + 1) = myList(place)
                    myList(place) = temp
                    place = place - 1
                Loop
                myList(place + 1) = myKey
            End If
        Next

        'output the sorted array
        For index = 0 To myList.Length - 1
            Console.Write(myList(index) & " ")
        Next

        Console.ReadKey() 'wait for keypress
    End Sub
End Module
```

19.1.3 Understanding and using Abstract Data Types (ADTs)

- **ADTs** is a **collection of data** and a **set of operations** on that data.
- There are several operations tha are essential when using an ADT:
  - **finding** an item already stored
  - **adding** a new item
  - **deleting** an item

- **Stack**
  - A list containing several items operating on the last in, first out (**LIFO**) principle.
  - Items can be **added** to the stack (**push**) and **removed** from the stack (**pop**).
  - The **first item added** to a stack is the **last item to be removed** from the stack.

- **Queue**
  - A list containing several items operating on the first in, first out (**FIFO**) principle.
  - Items can be **added** to the queue (**enqueue**) and **removed** from the queue (**dequeue**).
  - The **first item added** to a queue is the **first item to be removed** from the queue.

- **Linked list**
  - A list containing several items in which **each item in the list points to the next item in the list**.
  - In a linked list a **new item** is **always added** to the **start of the list**.

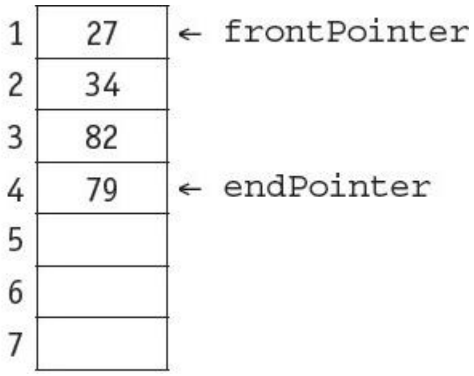
- **Stacks**, **queues** and **linked lists** all make use of **pointers** to manage their **operations**.
- **Items** stored in **stacks** and **queues** are **always added at the end**.
- **Linked lists** make use of an **ordering algorithm** for the items, often **ascending** or **descending**.

Stack

7		
6		
5		
4	79	← topPointer
3	82	
2	34	
1	27	← basePointer

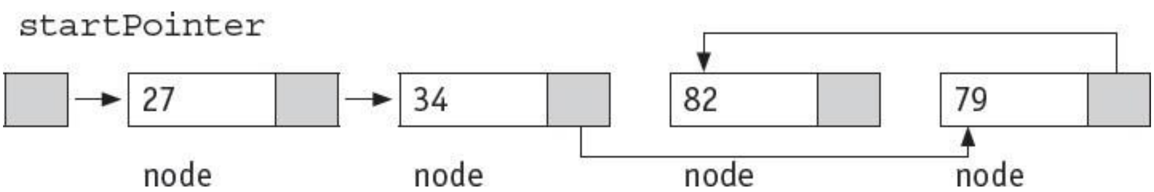
- A **stack** uses **two pointers**:
  - a **base pointer** points to the **first item** in the stack
  - a **top pointer** points to the **last item** in the stack.
- When they are **equal** there is **only one item** in the stack.

Queue



- A **queue** uses **two pointers**:
  - a **front pointer** points to the **first item** in the queue
  - a **rear pointer** points to the **last item** in the queue.
- When they are **equal** there is **only one item** in the queue.

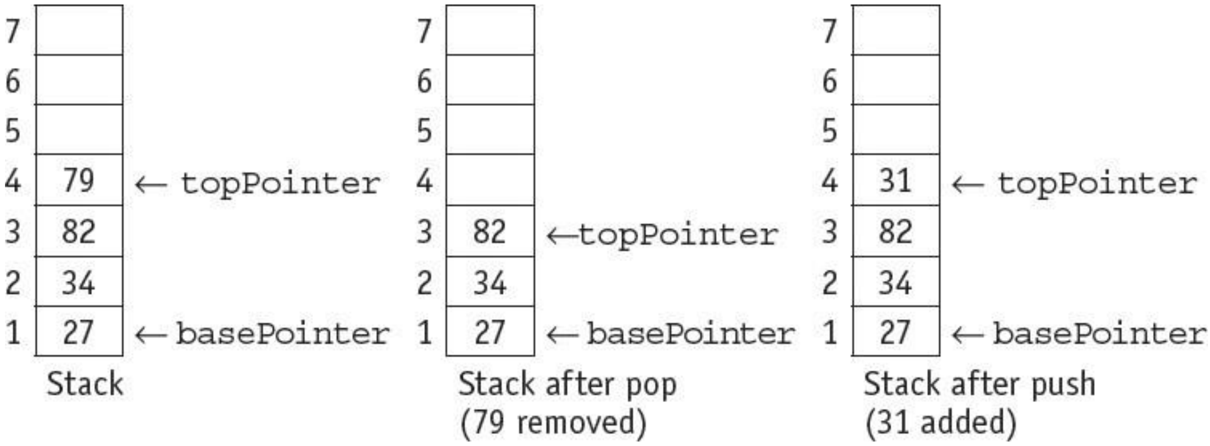
Linked List



- A **linked list** uses a **start pointer** that points to the **first item** in the **linked list**.
- Every **item** in a linked list is **stored together** with a **pointer** to the **next item**.
- This is called a **node**.
- The **last item** in a linked list has a **null pointer**.

Stack Operations

- The value of the basePointer always remains the same during stack operations:



- A **stack** can be **implemented** using an **array** and a **set of pointers**.
- As an **array** has a **finite size**, the stack may become **full** and this **condition** must be **allowed for**.

To set up a stack

```
DECLARE stack ARRAY[1:10] OF INTEGER
DECLARE topPointer : INTEGER
DECLARE basePointer : INTEGER
DECLARE stackful : INTEGER
basePointer ← 1
topPointer ← 0
stackful ← 10
```

To push an item, stored in item, onto a stack

```
IF topPointer < stackful
  THEN
    topPointer ← topPointer + 1
    stack[topPointer] ← item
  ELSE
    OUTPUT "Stack is full, cannot push"
ENDIF
```


To pop an item, stored in item, from the stack

```
IF topPointer = basePointer - 1
  THEN
    OUTPUT "Stack is empty, cannot pop"
  ELSE
    Item ← stack[topPointer]
    topPointer ← topPointer - 1
ENDIF
```

Stack Data Structure

```
Public Dim stack() As Integer = {Nothing, Nothing, Nothing, Nothing,
Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing}
Public Dim basePointer As Integer = 0
Public Dim topPointer As Integer = -1
Public Const stackFull As Integer = 10
Public Dim item As Integer
```

Stack pop operation

```
Sub pop()
      If topPointer = basePointer - 1 Then
            Console.WriteLine("Stack is empty, cannot pop")
      Else
            item = stack(topPointer)
            topPointer = topPointer - 1
      End If
End Sub
```

Stack push operation

```
Sub push(ByVal item)
      If topPointer < stackFull - 1 Then
      topPointer = topPointer + 1
            stack(topPointer) = item
      Else
            Console.WriteLine("Stack is full, cannot push")
      End if
End Sub
```

ACTIVITY 19E

- In your chosen programming language, write a program using subroutines to implement a stack with 10 elements.
- Test your program
  - by pushing two integers 7 and 32 onto the stack,
  - popping these integers off the stack,
  - then trying to remove a third integer,
  - and by pushing the integers 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10 onto the stack,
  - then trying to push 11 on to the stack.

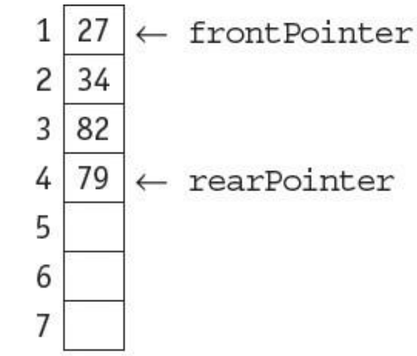
```
'VB program for stack
Module Module1
        Public stack() As Integer = {Nothing, Nothing, Nothing, Nothing,
                                Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing}
        Public basePointer As Integer = 0
        Public topPointer As Integer = -1
        Public Const stackFull As Integer = 10
        Public item As Integer

        Sub Main()
                push(7)
                push(32)
                pop()
                Console.WriteLine(item)
                pop()
                Console.WriteLine(item)
                pop()
                push(1)
                push(2)
                push(3)
                push(4)
                push(5)
                push(6)
                push(7)
                push(8)
                push(9)
                push(10)
                push(11)
                Console.ReadKey() 'wait for keypress
        End Sub

        Sub pop()
                If topPointer = basePointer - 1 Then
                        Console.WriteLine("Stack is empty, cannot pop")
                Else
                        item = Stack(topPointer)
                        topPointer = topPointer - 1
```



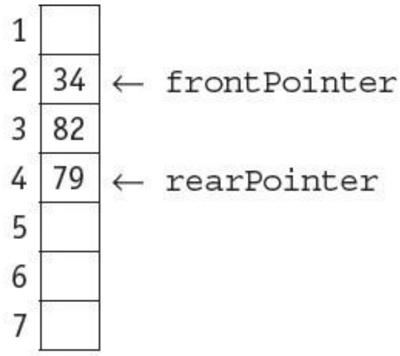
```
        End If
    End Sub

    Sub push(ByVal item)
        If topPointer < stackFull - 1 Then
            topPointer = topPointer + 1
            Stack(topPointer) = item
        Else
            Console.WriteLine("Stack is full, cannot push")
        End If
    End Sub
End Module
```

Queue Operations

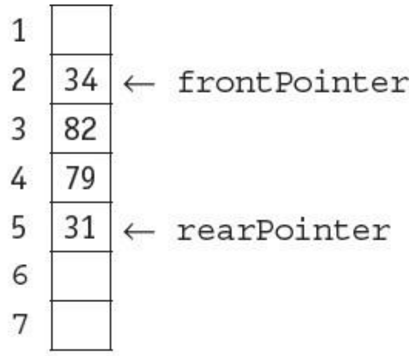
- The **value** of the **frontPointer** changes **after dequeue** but the **value** of the **rearPointer** changes **after enqueue**:



Queue

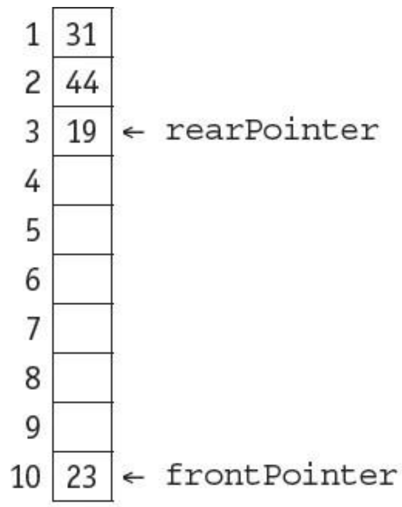


Queue after dequeue (27 removed)

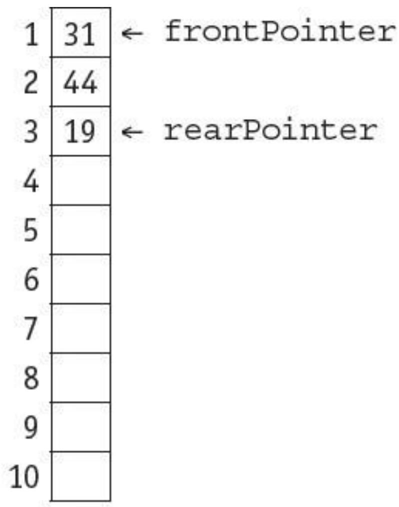


Queue after enqueue (31 added)

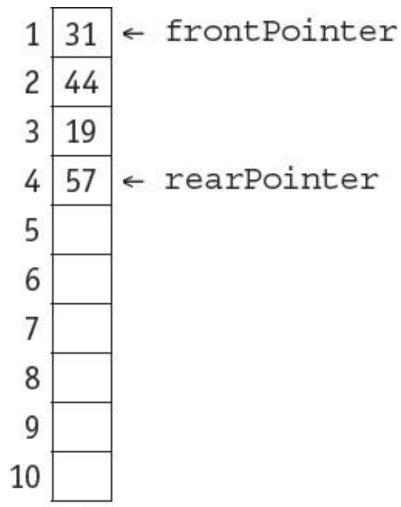
- A **queue** can be implemented using an **array** and a set of **pointers**.
- As an array has a **finite size**, the queue may become **full** and this condition must be **allowed for**.
- Also, as **items** are **removed** from the **front** and **added** to the **end** of a **queue**, the **position** of the **queue** in the array **changes**.
- Therefore, the **queue** should be **managed** as a **circular queue** to avoid moving the **position** of the **items** in the array **every time** an item is **removed**.



Queue length = 4



Queue length = 3 after dequeue (23 removed)



Queue length = 4 after enqueue (57 added)

- When a **queue** is implemented using an **array** with a **finite number** of **elements**, it is managed as a **circular queue**.
- Both pointers, **frontPointer** and **rearPointer**, are **updated** to point to the **first element** in the array (**lower bound**) after an operation where that pointer was originally pointing to the **last element** of the array (**upper bound**), providing the **length** of the **queue** does **not exceed the size of the array**.

To set up a queue

```
DECLARE queue ARRAY[1:10] OF INTEGER
DECLARE rearPointer : INTEGER
DECLARE frontPointer : INTEGER
DECLARE queueful : INTEGER
DECLARE queueLength : INTEGER
frontPointer ← 1
endPointer ← 0
upperBound ← 10
queueful ← 10
queueLength ← 0
```

To add an item, stored in item, onto a queue

```
IF queueLength < queueful
  THEN
    IF rearPointer < upperBound
      THEN
        rearPointer ← rearPointer + 1
      ELSE
        rearPointer ← 1
    ENDIF
    queueLength ← queueLength + 1
    queue[rearPointer] ← item
  ELSE
    OUTPUT "Queue is full, cannot enqueue"
ENDIF
```

To remove an item from the queue and store in item

```
IF queueLength = 0
  THEN
    OUTPUT "Queue is empty, cannot dequeue"
  ELSE
    Item ← queue[frontPointer]
    IF frontPointer = upperBound
      THEN
        frontPointer ← 1
      ELSE
        frontPointer ← frontPointer + 1
    ENDIF
    queueLength ← queueLength - 1
ENDIF
```

Queue data structure

```
Public Dim queue() As Integer = {Nothing,  Nothing, Nothing, Nothing,
Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing}
Public Dim frontPointer As Integer = 0
Public Dim rearPointer As Integer = -1
Public Const queueFull As Integer = 10
Public Dim queueLength As Integer =  0
Public Dim item As Integer
```

Queue enqueue (add item to queue) operation

```
Sub enQueue(ByVal item)
    If queueLength < queueFull Then
        If rearPointer < queue.length - 1 Then
            rearPointer = rearPointer + 1
        Else
            rearPointer = 0
        End If
        queueLength = queueLength + 1
        queue(rearPointer) = item
    Else
        Console.WriteLine("Queue is full, cannot enqueue")
    End If
End Sub
```

Queue dequeue (remove item from queue) operation

```
Sub deQueue()
    If queueLength = 0 Then
        Console.WriteLine("Queue is empty, cannot dequeue")
    Else
        item = queue(frontPointer)
        If frontPointer = queue.length - 1 Then
            frontPointer = 0
        Else
            frontPointer = frontPointer + 1
        End if
        queueLength = queueLength - 1
    End If
End Sub
```

ACTIVITY 19F

- In your chosen programming language, write a program using subroutines to implement a queue with 10 elements.
- Test your program
  - by adding two integers 7 and 32 to the queue,
  - removing these integers from the queue,
  - then trying to remove a third integer,
  - and by adding the integers 1, 2, 3, 4, 5, 6, 7, 8, 9 and 10 to the queue then trying to add 11 to the queue.

```
'VB program for queue
Module Module1
        Public Dim frontPointer As Integer = 0
        Public Dim rearPointer As Integer = -1
        Public Const queueFull As Integer = 10
        Public Dim queueLength As Integer = 0
        Public Dim item As Integer
        Public Dim queue() As Integer = {Nothing, Nothing, Nothing, Nothing, Nothing, Nothing,
                                         Nothing, Nothing, Nothing, Nothing, Nothing}

        Public Sub Main()
                Console.ReadKey()
                enQueue(7)
                enQueue(32)
                deQueue()
                Console.WriteLine(item)
                deQueue()
                Console.WriteLine(item)
                deQueue()
                Console.ReadKey()
                enQueue(1)
                enQueue(2)
                enQueue(3)
                enQueue(4)
                enQueue(5)
                enQueue(6)
                enQueue(7)
                enQueue(8)
                enQueue(9)
                enQueue(10)
                enQueue(11)
                Console.ReadKey()
        End Sub
```

```
    Sub enQueue(ByVal item)
        If queueLength < queueFull Then
            If rearPointer < queue.length - 1 Then
                rearPointer = rearPointer + 1
            Else
                rearPointer = 0
            End If

            queueLength = queueLength + 1
            queue(rearPointer) = item
        Else
            Console.WriteLine("Queue is full, cannot enqueue")
        End if
    End Sub

    Sub deQueue()
        If queueLength = 0 Then
            Console.WriteLine("Queue is empty, cannot dequeue")
        Else
            item = queue(frontPointer)
            If frontPointer = queue.length - 1 Then
                frontPointer = 0
            Else
                frontPointer = frontPointer + 1
            End if
            queueLength = queueLength - 1
        End If
    End Sub
End Module
```

Linked List Operations

- A **linked list** can be **implemented** using **two 1D arrays**, one for the **items** in the **linked list** and another for the **pointers** to the **next item** in the list, and a **set of pointers**.
- As an **array** has a **finite size**, the linked list may become **full** and this **condition** must be **allowed for**. Also, as items can be **removed** from **any position** in the linked list, the **empty positions** in the array must be **managed** as an **empty linked list**, usually called the **heap**.

- The following diagrams demonstrate the operations of linked lists.
- The **startPointer = –1**, as the list has **no elements**. The **heap** is set up as a linked list **ready for use**.

Index	Element		Index	Pointer
0		← heapPointer	0	1
1			1	2
2			2	3
3			3	4
4			4	5
5			5	6
6			6	7
7			7	8
8			8	9
9			9	10
10			10	11
11			11	–1

Empty linked list elements — Empty linked list pointers

- The **startPointer** is set to the **element** pointed to by the **heapPointer** where **37** is **inserted**.
- The **heapPointer** is set to point to the **next element** in the **heap** by using the **value** stored in the **element** with the **same index** in the **pointer list**.
- Since this is also the **last element** in the list the node pointer for it is **reset to –1**.

Index	Element		Index	Pointer
0	37	← startPointer	0	–1
1		← heapPointer	1	2
2			2	3
3			3	4
4			4	5
5			5	6
6			6	7
7			7	8
8			8	9
9			9	10
10			10	0
11			11	–1

Linked list with element 37 added — Linked list pointers with one element 37 added

- The **startPointer** is changed to the **heapPointer** and **45** is stored in the **element indexed** by the **heapPointer**. The **node pointer** for this **element** is set to the **old startPointer**.
- The **node pointer** for the **heapPointer** is **reset** to point to the **next element** in the **heap** by using the **value** stored in the **element** with the **same index** in the **pointer list**.

0	37	
1	45	← startPointer
2		← heapPointer
3		
4		
5		
6		
7		
8		
9		
10		
11		

Linked list with element 37 then 45 added

0	–1
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10
10	0
11	–1

Linked list pointers with element 37 then 45 added

- The **process** is **repeated** when **12** is **added** to the **list**.

0	37	
1	45	
2	12	← startPointer
3		← heapPointer
4		
5		
6		
7		
8		
9		
10		
11		

Linked list with elements 37, 45 then 12 added

0	–1
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9
9	10
10	0
11	–1

Linked list pointers with elements 37, 45 then 12 added

To set up a linked list

```
DECLARE mylinkedList ARRAY[0:11] OF INTEGER
DECLARE myLinkedListPointers ARRAY[0:11] OF INTEGER
DECLARE startPointer : INTEGER
DECLARE heapStartPointer : INTEGER
DECLARE index : INTEGER
heapStartPointer ← 0
startPointer ← -1 // list empty
FOR index ← 0 TO 11
    myLinkedListPointers[index] ← index + 1
NEXT index
// the linked list heap is a linked list of all the
spaces in the linked list, this is set up when the
linked list is initialised
myLinkedListPointers[11] ← -1
// the final heap pointer is set to -1 to show no
further links
```

Identifier	Description
myLinkedList	Linked list to be searched
myLinkedListPointers	Pointers for linked list
startPointer	Start of the linked list
heapStartPointer	Start of the heap
index	Pointer to current element in the linked list

- The **table** below shows an **empty linked list** and its corresponding **pointers**.

		myLinkedList	myLinkedListPointers
heapstartPointer	[0]		1
	[1]		2
	[2]		3
	[3]		4
	[4]		5
	[5]		6
	[6]		7
	[7]		8
	[8]		9
	[9]		10
	[10]		11
	[11]		–1

startPointer = -1

Finding an item in a linked list

		myLinkedList	myLinkedListPointers
	[0]	27	–1
	[1]	19	0
	[2]	36	1
	[3]	42	2
startPointer →	[4]	16	3
heapStartPointer →	[5]		6
	[6]		7
	[7]		8
	[8]		9
	[9]		10
	[10]		11
	[11]		–1

- The algorithm to **find if an item** is in the linked list **myLinkedList** and **return** the **pointer** to the **item if found** or a **null pointer** if **not found**, could be written as a **function** in pseudocode as shown below:

```
DECLARE itemSearch : INTEGER
DECLARE itemPointer : INTEGER
CONSTANT nullPointer = -1
FUNCTION find(itemSearch) RETURNS INTEGER
DECLARE found : BOOLEAN
itemPointer ← startPointer
found ← FALSE
    WHILE (itemPointer <> nullPointer) AND NOT found DO
        IF myLinkedList[itemPointer] = itemSearch
          THEN
            found ← TRUE
          ELSE
            itemPointer ← myLinkedListPointers[itemPointer]
        ENDIF
    ENDWHILE
RETURN itemPointer
// this function returns the item pointer of the value found or -1 if the
item is not found
```


- The following programs use a function to **search for an item** in a populated linked list:

```
'VB program for finding an item in a linked list
Module Module1
    Public Dim startPointer As Integer = 4
    Public Const nullPointer As Integer = -1
      Public Dim item As Integer
      Public Dim itemPointer As Integer
      Public Dim result As Integer
      Public Dim myLinkedList() As Integer = {27, 19, 36, 42, 16,
        Nothing, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing}
      Public Dim myLinkedListPointers() As Integer = {-1, 0, 1, 2,
        3, 6, 7, 8, 9, 10, 11, -1}
         Public Sub Main()
         'enter item to search for
            Console.Write("Please enter item to be found ")
            item = Integer.Parse(Console.ReadLine())
            result = find(item)
            If result <> -1 Then
               Console.WriteLine("Item found")
            Else
               Console.WriteLine("Item not found")
            End If
            Console.ReadKey()
         End Sub

    Function find(ByVal itemSearch As Integer) As Integer
         Dim found As Boolean = False
         itemPointer = startPointer
         While (itemPointer <> nullPointer) And Not found
            If itemSearch = myLinkedList(itemPointer) Then
            found = True
          Else
            itemPointer = myLinkedListPointers(itemPointer)
          End If
         End While
         Return itemPointer
    End Function
End Module
```

Populating the linked list

Calling the find function

Defining the find function

Inserting items into a linked list

```
DECLARE itemAdd : INTEGER
DECLARE startPointer : INTEGER
DECLARE heapstartPointer : INTEGER
DECLARE tempPointer : INTEGER
CONSTANT nullPointer = -1
PROCEDURE linkedListAdd(itemAdd)
  // check for list full
  IF heapStartPointer = nullPointer
    THEN
      OUTPUT "Linked list full"
    ELSE
      // get next place in list from the heap
      tempPointer ← startPointer // keep old start pointer
      startPointer ← heapStartPointer  // set start pointer to next position in heap
      heapStartPointer ← myLinkedListPointers[heapStartPointer] // reset heap start pointer
      myLinkedList[startPointer] ← itemAdd // put item in list
      myLinkedListPointers[startPointer] ← tempPointer  // update linked list pointer
  ENDIF
ENDPROCEDURE
```

Identifier	Description
startPointer	Start of the linked list
heapStartPointer	Start of the heap
nullPointer	Null pointer set to -1
itemAdd	Item to add to the list
tempPointer	Temporary pointer

- Below shows the populated linked list and its corresponding pointers again:

		myLinkedList	myLinkedListPointers
	[0]	27	−1
	[1]	19	0
	[2]	36	1
	[3]	42	2
startPointer →	[4]	16	3
heapStartPointer →	[5]		6
	[6]		7
	[7]		8
	[8]		9
	[9]		10
	[10]		11
	[11]		−1

- The linked list, myLinkedList, will now be as shown below:

		myLinkedList	myLinkedListPointers
	[0]	27	–1
	[1]	19	0
	[2]	36	1
	[3]	42	2
	[4]	16	3
startPointer →	[5]	18	4
heapStartPointer →	[6]		7
	[7]		8
	[8]		9
	[9]		10
	[10]		11
	[11]		–1

- The following procedure adds an item to a linked list:

```
Sub insert (ByVal itemAdd)
    Dim tempPointer As Integer
    If heapStartPointer = nullPointer Then
       Console.WriteLine("Linked List full")
    Else
       tempPointer = startPointer          ← Adjusting the pointers and adding the item
       startPointer = heapStartPointer
       heapStartPointer = myLinkedListPointers(heapStartPointer)
       myLinkedList(startPointer) = itemAdd
       myLinkedListPointers(startPointer) = tempPointer
    End if
End Sub
```

Deleting items from a linked list

```
DECLARE itemDelete : INTEGER
DECLARE oldIndex : INTEGER
DECLARE index : INTEGER
DECLARE startPointer : INTEGER
DECLARE heapStartPointer : INTEGER
DECLARE tempPointer : INTEGER
CONSTANT nullPointer = -1
PROCEDURE linkedListDelete(itemDelete)
  // check for list empty
  IF startPointer = nullPointer
    THEN
      OUTPUT "Linked list empty"
    ELSE
      // find item to delete in linked list
      index ← startPointer
      WHILE myLinkedList[index] <> itemDelete AND
        (index <> nullPointer) DO
          oldIndex ← index
          index ← myLinkedListPointers[index]
      ENDWHILE
      IF index = nullPointer
        THEN
          OUTPUT "Item ", itemDelete, " not found"
        ELSE
        // delete the pointer and the item
         tempPointer ← myLinkedListPointers[index]
         myLinkedListPointers[index] ← heapStartPointer
         heapStartPointer ← index
         myLinkedListPointers[oldIndex] ← tempPointer
     ENDIF
  ENDIF
ENDPROCEDURE
```

Identifier	Description
startPointer	Start of the linked list
heapStartPointer	Start of the heap
nullPointer	Null pointer set to −1
index	Pointer to current list element
oldIndex	Pointer to previous list element
itemDelete	Item to delete from the list
tempPointer	Temporary pointer

- The trace table below shows the algorithm being used to delete 36 from myLinkedList.

startPointer	heapStartPointer	itemDelete	index	oldIndex	tempPointer
Already set to 4	Already set to 5	36	4	4	
			3	3	
			2		
	2				1

- The linked list, myLinkedList, will now be as follows.

		myLinkedList	myLinkedListPointers
	[0]	27	−1
	[1]	19	0
heapStartPointer →	[2]	36	6
	[3]	42	1
	[4]	16	3
startPointer →	[5]	18	4
	[6]		7
	[7]		8
	[8]		9
	[9]		10
	[10]		11
	[11]		−1

updated pointers


```
Sub delete (ByVal itemDelete)
    Dim tempPointer, index, oldIndex  As Integer
    If startPointer = nullPointer Then
        Console.WriteLine("Linked List empty")
    Else
        index = startPointer
        While myLinkedList(index) <> itemDelete And index <> nullPointer
            Console.WriteLine( myLinkedList(index) & " " & index)
            Console.ReadKey()
            oldIndex = index
            index = myLinkedListPointers(index)
        End While
        if index = nullPointer Then
            Console.WriteLine("Item " & itemDelete & " not found")
        Else
            myLinkedList(index) = nothing
            tempPointer = myLinkedListPointers(index)
            myLinkedListPointers(index) = heapStartPointer
            heapStartPointer = index
            myLinkedListPointers(oldIndex) = tempPointer
        End If
    End If
End Sub
```


ACTIVITY 19G

- In the programming language of your choice, use the code given to write a program to set up the populated linked list and find an item stored in it.

ACTIVITY 19H

- Use the algorithm to add 25 to myLinkedList. Show this in a trace table and show myLinkedList once 25 has been added. Add the insert procedure to your program, add code to input an item, add this item to the linked list then print out the list and the pointers before and after the item was added.

ACTIVITY 19I

- Use the algorithm to remove 16 from myLinkedList. Show this in a trace table and show myLinkedList once 16 has been removed. Add the delete procedure to your program, add code to input an item, delete this item to the linked list, then print out the list and the pointers before and after the item was deleted

```
'VB program for a linked list
Module Module1
    Public Dim As Integer heapStartPointer = 5
    Public Dim As Integer startPointer= 4
    Public Const As Integer nullPointer = -1
    Public Dim item As Integer
    Public Dim index As Integer
    Public Dim itemPointer As Integer
    Public Dim result As Integer
    Public Dim myLinkedList() As Integer = {27, 19, 36, 42, 16, Nothing, Nothing, Nothing, Nothing, Nothing, Nothing,
                                                                                                        Nothing}

    Public Dim myLinkedListPointers() As Integer = {-1, 0, 1, 2, 3, 6, 7, 8, 9, 10, 11, -1}

    Public Sub Main()
        'enter item to delete
        Console.Write("Please enter item to remove from list ")
        item = Integer.Parse(Console.ReadLine())
        delete(item)
        For index = 0 To myLinkedList.Length - 1
            Console.Write(myLinkedList(index) & " ")
        Next
        Console.WriteLine()

        For index = 0 To myLinkedListPointers.Length - 1
            Console.Write(myLinkedListPointers(index) & " ")
        Next
        Console.WriteLine()

        'enter item to insert
        Console.Write("Please enter item to add to list ")
        item = Integer.Parse(Console.ReadLine())
        insert(item)
```



```
        For index = 0 To myLinkedList.Length - 1
            Console.Write(myLinkedList(index) & " ")
        Next

        For index = 0 To myLinkedListPointers.Length - 1
            Console.Write(myLinkedListPointers(index) & " ")
        Next

        'enter item to search for
        Console.Write("Please enter item to be found ")
        item = Integer.Parse(Console.ReadLine())
        result = find(item)

        If result <> -1 Then
            Console.WriteLine("Item found")
        Else
            Console.WriteLine("Item not found")
        End If
        Console.ReadKey()
End Sub

Sub insert (ByVal itemAdd)
    Dim tempPointer As Integer
    If heapStartPointer = nullPointer Then
        Console.WriteLine("Linked List full")
    Else
        tempPointer = startPointer
        startPointer = heapStartPointer
        myLinkedList(startPointer) = itemAdd
        myLinkedListPointers(startPointer) = tempPointer
    End if
End Sub

Sub delete (ByVal itemDelete)
    Dim tempPointer, index, oldIndex As Integer

    If startPointer = nullPointer Then
        Console.WriteLine("Linked List empty")
    Else
        index = startPointer
        While myLinkedList(index) <> itemDelete And index <> nullPointer
            Console.WriteLine( myLinkedList(index) & " " & index)
            Console.ReadKey()
            oldIndex = index
            index = myLinkedListPointers(index)
        End While

        If index = nullPointer Then
            Console.WriteLine("Item " & itemDelete & " not found")
        Else
            myLinkedList(index) = nothing
            tempPointer = myLinkedListPointers(index)
            myLinkedListPointers(index) = heapStartPointer
            heapStartPointer = index
```



```
                    myLinkedListPointers(oldIndex) = tempPointer
            End if
    End If
End Sub

Function find (ByVal itemSearch As Integer) As Integer
    Dim Found = False As Boolean
    itemPointer = startPointer
    While (itemPointer <> nullPointer) And Not found
            If itemSearch = myLinkedList(itemPointer) Then
                    found = True
            Else
                    itemPointer = myLinkedListPointers(itemPointer)
            End if
    End While
    Return itemPointer
End Function

End Module
```