DATA STRUCTURES

COMPUTER LANGUAGES WILL HABE **BUILT-IN ELEMENTARY DATA TYPES** SUCH AS **INTEGER, REAL, BOOLEAN AND CHAR**. THEY ALSO HAVE SOME BUILT-IN **STRUCTURED DATA TYPES** SUCH AS: **STRING, ARRAY AND RECORD**.

THESE ARE MADE UP OF A NUMBER OF ELEMENTS OF A SPECIFIED ELEMENTARY DATA TYPE.

ARRAYS

#An array is a data structure that allows you to store multiple elements under the same common data type. These are static data structures and are immutable (can't change in size/be edited)

It is a finite, ordered set of elements of the **same data type**. Finite means there is a specific number of elements and ordered implies that there is a first, second and third element of the array. They are usually used when the size of the data structure is known in advanced e.g. a stack (although a stack can be both static and dynamic)

- **A 1-D array** is a linear set of elements of the same data type.
- **A 2-D array** is simply a list of lists. It is like having two axes (x,y). We may want to have an index position to indicate the element in the main list (this element will be a list in itself) and then a second index to indicate the precise element within that element's list.

  In exams, we commonly imagine **2-D arrays as a table** where there are **columns and rows** of elements. Elements in their array are referred to **by index of [row, column].**

  We write pseudocode of 2-D arrays as:

  ```
  Array = [[L1E1, L1E2, L1E3, L1E4],
           [L2E1, L2E2, L2E3, L2E4],
           [L3E1, L3E2, L3E3, L3E4]]
  ```

- **A 3-D array** = a set of elements of the same type, indexed by 3 integers. [x,y,z]
- Arrays can have n-dimensions and so indexed by n-integers.

LISTS

#A list = an **abstract data type** consisting of **a number of items** in which the **same item** may occur **more than once**. The list is s**equenced. A list is mutable and dynamic.**

The list is sequenced and so can refer to first, second, third,……. Item and we can also refer to the last element of a list.

List operation	Description	Example	List contents	Return Value or updates list
isEmpty(list)	Tests for empty list	a.isEmpty()	[2,3,7,1]	False
isFull(list)	Tests for full list if static	a.isFull()	[2,3, ,]	False
append(item)	Adds a new item to the rear (end) of the list	a.append(33)	[2,4,6]	[2,4,6,33]
remove(item)	Removes the first occurrence of an item form a list (decreases length of list by 1	a.remove(13)	[4,13,6,8,13]	[4,6,8,13]

List operation	Description	Example	List contents	Return Value or updates list
pop() or pop(index)	Removes and returns the last item in the list OR the item of stated index	a.pop()	[2,3,7,1]	[2,3,7] & returns 1
.index(item)	Searches for an item in list and returns index	a.index(22)	[33,44,22,11]	2
search(item)	Searches for an item in list and returns Boolean	a.search(22)	[33,44,22,11]	True
length() or length(list)	Returns integer value of length of list	a.length() or length(a)	[33,44,22,11]	4
insert(pos,item)	Inserts new item at position and shifts everything else along to right (increases list length)	a.insert(2,7)	[45,18,3,13,33]	[45,18,7,3,13,33]

LISTS AND TUPLES

These are python's versions of arrays.

- **Tuples** are lists that cannot be edited. Once created it cannot be changed in an way. This property is known **as immutable** (can't be edited). Can hold multiple data types.
- **Lists** are arrays that can be edited after creation so are **mutable**. Items, therefore, can be added or deleted. However, they only hold one data type.

This means that the sizes of lists can grow, they are **dynamic** unlike **an array or a tuple** which will ALWAYS have a **fixed size so can't grow**. **A tuple is static.**

Lists use square brackets [,] **vs** **Tuples use round brackets (,)**
Mutable **Immutable**
More strenuous on memory resources **Less strenuous on memory resources**

Using tuples in the correct places will therefore gain a small performance increase.

DYNAMIC VS STATIC DATA STRUCTURE

A dynamic structure refers to a **collection of data in memory** that has the **ability to grow or shrink in size**. It does this with the **aid of the heap**, a **portion of memory** from which **free space** is automatically **allocated or de-allocated as required.**

A potential **draw back** of a **dynamic data structure** (like **lists** in python) is that an **overflow error** may occur when the **structure exceeds its maximum memory limit**.

Dynamic data structures are very useful when implementing data structures **do not have a finite size** (their maximum size is **not known in advance**). Good examples are **queues**, which are **dynamic** so can grow in size.

NOTE: The queue can be given some arbitrary maximum in advanced to prevent memory overflow but memory allocation in advanced is not required.

Another advantage of using a built-in dynamic data structure such as a list is that there are many methods or functions that can be used in the implementation of other data structures like stacks or queues.

e.g. length, pop, insert, remove, append and search.

A static data structure *such as an* ***array*** *(or tuple) is fixed in size (immutable). It cannot increase in size nor free up space while it is running. This is more demanding on memory resources.*

The disadvantage of using an array to implement a dynamic data structure is that the size of the array must be declared in advance. This is not suitable for queues since a number of items fills up the array and no more can be added, regardless of how much frees space there is in memory.

RECORDS

#A record is a data structure used to group together a collection of related fields under a single name. This may contain different data types.

When using records we must take a 3 step process:

1. **Define** the record structure: **what fields** will be in the record.
2. **Declare a variable** or array from the **variable record.**
3. **Assign and retrieve** data from the **variable record.**

An example in visual basic

Record Structures

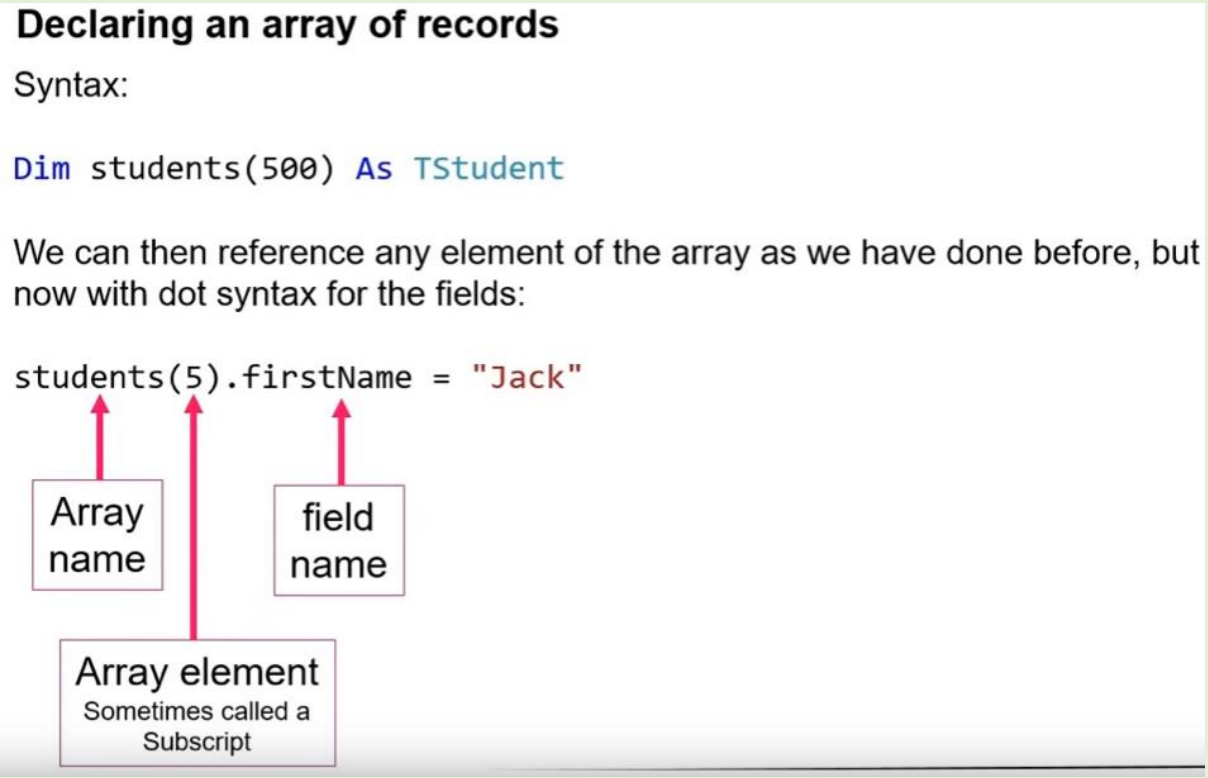
A complete example:

```
Module Module1
    Structure TStudent
        Dim firstName As String
        Dim surname As String
        Dim depositPaid As Double
        Dim datePaid As Date
    End Structure

    Sub Main()
        Dim Student1 As TStudent

        Student1.firstName = "Jeff"
        Student1.surname = "Williams"
        Student1.depositPaid = 36.0
    End Sub
End Module
```

1. Defining the record structure
2. Declaring the variables in the record for the fields (this is also declaring what fields there will be in VB)
3. Assigning and retrieving data from the variable record

RECORDS ARE NTO VERY USEFUL IF THEY CAN ONLY STORE DETAILS ABOUT ONE ELEMENT.
WE OFTEN USE ARRAYS TOGETHER WITH A RECORD STRUCTURE.
THIS ALLOWS US TO ALSO GET AROUND THE LIMITATION THAT ARRAYS CAN ONLY STORE ONE DATA TYPE.

We use an array to store elements that are part of the record data structure. So rather than separating the variables Student1 Student2 Student3 Student4 (which are all storing data in the record structure TStudent), we can instead have an array Students[] that has multiple items all using a record.



QUEUES

THE DATA STRUCTURE

#A queue is a data structure operating on a FIFO basis (First in first out), containing an ordered collection of items. New items may only be added to the rear of the queue and elements may only be retrieved from the front of the queue.

Items are **enqueued** (added) or **dequeued** (removed).

The sequence of items is defined by the order in which they are added. **The size of a queue depends on the number of items in it**, just like a queue at traffic lights.

Operations on queues: **A queue is a dynamic and mutable structure.**

In exams, the front pointer is made to point at the front (where items are removed) while the rear pointer points at the back (where items are added).

The following queue operations are needed:

- **enQueue(item)** = Add a new item to the rear of the queue
- **deQueue()** = Remove the front item from the queue and return it
- **isEmpty()** = Test to see whether the queue is empty
- **isFull()** = Test to see whether the queue is full

APPLICATIONS OF QUEUES

1. **Outputs waiting to be printed** are commonly stored in a **queue on disk**. For example, on a network, numerous clients may send work documents to a print server at a similar time. By putting the output into a **queue on disk**, the output is printed on a **first come first serve** basis as soon as the Printer is free.
2. **Characters** typed into a keyboard are held in a **queue in a keyboard buffer**.
3. **Queues are useful in simulation programs**. A simulation program is one which attempts to model a real-life situation so as to learn something about it.

   An example of a program that simulates customers arriving at random times at the checkouts of supermarkets, and taking random times to pass through the checkout. With the aid of a simulation program, the optimum number of check-out counters can be established.

   The CPU scheduler often uses algorithms that use queues.

PSEUDOCODE FOR QUEUE (UNLIMITED LENGTH)

```
PROCEDURE QUEUE():

        queue = []
        top_pointer = 0

        rear_pointer = 0
        continue = "Y"
        WHILE continue == "Y":

                choice = int(input(Would you like to enqueue (1), dequeue (2) or view items (3)? "))
                IF choice == 1 THEN:
                        IF queue.IsFull() == TRUE THEN
                                print("Queue full")
                        ELSE:
                                 item = str(input("Enter item to enqueue: "))
                                queue.append(item)
                                top_pointer = top_pointer + 1

                ELSE IF choice == 2 THEN:
                        IF queue.IsEmpty = TRUE THEN
                                print("Queue is empty")
                        ELSE:
                                item = queue.remove(rear_pointer)
                                print("Item dequeued: ", item)
                                rear_pointer = rear_pointer + 1


                        END IF
                ELSE:
                        FOR i = 0 to len(queue – 1):
                                print("Item : ", queue[i] )
                        NEXT i
                END IF

                continue = str(input("Would you like to continue (y/n) ? "))
                continue = continue.upper()
        END WHILE

END PROCEDURE
```

PSEUDOCODE FOR QUEUE (LIMITED LENGTH) – VERY VERY UNLIKELY

```
PROCEDURE QUEUE():

    queue = [“”] * 10
    top_pointer = 0


    continue = “Y”
    WHILE continue == “Y”:


        choice = int(input(“Would you like to enqueue (1)  or dequeue (2)  or view items (3) ? “))
        IF choice == 1 THEN
            IF top_pointer == len(queue) THEN
                print(“The queue is full”)
            ELSE:
                item = str(input(“Enter item to enqueue: “))
                queue[top_pointer] = item
                top_pointer = top_pointer + 1
            END IF
        ELSE IF choice == 2 THEN
            IF top_pointer == 0 THEN
                print(“The queue is empty”)
            ELSE:
                item = queue[0]
                print(“Item dequeued: “, item)
                FOR i = 0 to len(queue) – 2:
                    queue[i] = queue[i + 1]
                    queue[i+1] = “”
                NEXT i
                top_pointer = top_pointer – 1


            END IF
        ELSE:
            FOR i = 0 to len(queue – 1):
                print(“Item: “, queue[i] )
            NEXT i
        END IF

        continue = str(input(“Would you like to continue? (y/n) “))
        continue = continue.upper()

    END WHILE

END PROCEDURE
```

PYTHON QUEUE ALGORITHM (LIMITTED LENGTH) – VERY VERY UNLIKELY

```
#Queue data structure: Adam Suttle

#The algortihm is identical to the stack algorithm only when dequeuing (popping) we remove the first item
#this means we then have to shift all elements to the left one place including the top pointer


def queue_proc():
    queue = [""] * 10
    top_pointer = 0

#-----------------------------------------------------------------------------------------------------------------------------------------
    response = "Y"
    while response == "Y":
#-----------------------------------------------------------------------------------------------------------------------------------------

        choice = int(input("Would you like to dequeue (1) or enqueue (2) or view items(3) from the the queue? "))

        if choice == 1:                                                         #Dequeue items from start of queue

            if top_pointer == 0:                                                #No items can be dequeued when empty
                print("The queue is empty")
            else:
                item = queue[0]                                                 #Dequeues front item
                print("Item dequeued: ", item)

                for i in range (0,(len(queue)-1)):
                    queue[i] = queue[i + 1]                                     #Updates front of queue using shifting
                    queue[i + 1] = ""

                top_pointer -= 1                                                #Updates back of queue


        elif choice == 2:                                                        #Enqueue item to the back of the queue

            if top_pointer == len(queue):                                        #If the pointer is greater than last index (>9)
                print("The queue is full")
            else:

                item = str(input("Enter an item to enqueue to the queue: "))
                queue[top_pointer] = item                                       #Adds new item to back of queue
                top_pointer += 1                                                #Incremeents the pointer by 1

        else:
            for i in range (0, len(queue)):                                     #Displays every item in the queue
                print("Item ", i, ": ", queue[i])

#-----------------------------------------------------------------------------------------------------------------------------------------
    response = str(input("\nContinue? (y/n)"))
    response = response.upper()
#-----------------------------------------------------------------------------------------------------------------------------------------

queue_proc()
```

Example of this program in use

```
Would you like to dequeue (1) or enqueue (2) or view items(3) from the the queue? 2
Enter an item to enqueue to the queue: 1
Would you like to dequeue (1) or enqueue (2) or view items(3) from the the queue? 2
Enter an item to enqueue to the queue: 3
Would you like to dequeue (1) or enqueue (2) or view items(3) from the the queue? 2
Enter an item to enqueue to the queue: 5
Would you like to dequeue (1) or enqueue (2) or view items(3) from the the queue? 3
Item  0 :  1
Item  1 :  3
Item  2 :  5
Item  3 :
Item  4 :
Item  5 :
Item  6 :
Item  7 :
Item  8 :
Item  9 :
Would you like to dequeue (1) or enqueue (2) or view items(3) from the the queue? 1
Item dequeued:  1
Would you like to dequeue (1) or enqueue (2) or view items(3) from the the queue? 1
Item dequeued:  3
Would you like to dequeue (1) or enqueue (2) or view items(3) from the the queue? 3
Item  0 :  5
Item  1 :
Item  2 :
Item  3 :
Item  4 :
Item  5 :
Item  6 :
Item  7 :
Item  8 :
Item  9 :
Would you like to dequeue (1) or enqueue (2) or view items(3) from the the queue?
```

```
def proc_queue2():
    queue = []
    top_pointer = 0

    response = "Y"
    while response == "Y":

        choice = int(input("Would you like to enqueue (1), dequeue (2) or view items (3)? "))

        if choice == 1:
            item = str(input("Enter item to enqueue: "))
            queue.append(item)
            top_pointer = top_pointer + 1

        elif choice == 2:
            if top_pointer == 0:
                print("List is empty")
            else:
                item = queue[0]
                print("Item dequeued: ", item)

                for i in range (0, len(queue) -1):
                    queue[i] = queue[i +1]

                top_pointer = top_pointer -1
                queue.pop(top_pointer -1)
        else:
            for i in range(0, len(queue)):
                print("Item ", i, ": ", queue[i] )


    response = str(input("WOuld you like to continue (y/n): "))
    response = response.upper()

proc_queue2()
```

Example of this program in use

```
Would you like to enqueue (1), dequeue (2) or view items (3)? 1
Enter item to enqueue: 1
Would you like to enqueue (1), dequeue (2) or view items (3)? 1
Enter item to enqueue: 2
Would you like to enqueue (1), dequeue (2) or view items (3)? 1
Enter item to enqueue: 3
Would you like to enqueue (1), dequeue (2) or view items (3)? 1
Enter item to enqueue: 4
Would you like to enqueue (1), dequeue (2) or view items (3)? 3
Item  0 :  1
Item  1 :  2
Item  2 :  3
Item  3 :  4
Would you like to enqueue (1), dequeue (2) or view items (3)? 2
Item dequeued:  1
Would you like to enqueue (1), dequeue (2) or view items (3)? 3
Item  0 :  2
Item  1 :  3
Item  2 :  4
Would you like to enqueue (1), dequeue (2) or view items (3)? 2
Item dequeued:  2
Would you like to enqueue (1), dequeue (2) or view items (3)? 3
Item  0 :  3
Item  1 :  4
Would you like to enqueue (1), dequeue (2) or view items (3)? 1
Enter item to enqueue: 5
Would you like to enqueue (1), dequeue (2) or view items (3)? 3
Item  0 :  3
Item  1 :  4
Item  2 :  5
Would you like to enqueue (1), dequeue (2) or view items (3)?
```

What are the differences:

The queue algorithm is the same as the stack algorithm for enqueueing/pushing. However, for dequeuing (popping), the queue will remove the item from the index position 0 rather than the top position.

The queue will then also have to shift all the elements one place to the left and replace the element in front with a blank at each stage to remove the duplicate at the end.

The top pointer will then decrease

When dealing with unlimited length. We must append items to the end of the list and then increase the top pointer by 1 (both stacks and queues). We still need the top_poitner but not for directing the insertion of a new item.

When dequeuing from an unlimited queue, we must remove the first item (index option 1) then shift all elements to the left one but we do NOT replace the item ahead with a blank since the last item in the list (which is a duplicate) needs to be entirely deleted. We delete the last index position indicated by (top_poistion -1).