Section 7 - Data Structures

Storing "lists" of data

A static data structure is fixed in size.

A dynamic data structure has a variable size. It does this with the use of a heap, which is a portion of memory from which space is automatically allocated and deallocated as required.

Arrays

Arrays are static data structures, where items are all of the same type and accessed via indices. A zero-indexed array means the first item is accessed with the index 0.

A two-dimensional (2D) array is where each item is a 1D array. This can be visualised as a table. Individual items are referred to as "array[row, column]" in OCR reference language. You can have more than two dimensions for an array.

Tuples

Tuples behave and are accessed like an array. The difference is that tuples are immutable, meaning their values cannot be changed, and tuples can hold data of more than one type.

Records

[Links to databases, strikingly similar to attributes in OOP]

Best for data storage. Each record has multiple fields, each of which hold one data type. They are defined, declared and accessed as follows in OCR reference language:

```
// Define
record1 = record
  field1
  field2
  field3
endrecord
```

```
// Declare
examplerecord : record1

// Access
examplerecord.field1 = "Hello World"
examplerecord.field2 = "Fizz"
examplerecord.field3 = "Buzz"
print(examplerecord.field1) // Hello World
```

Abstract data types

Data types created by the programmer rather than the language. A logical description of how the data is viewed and manipulated, but the user doesn't need to necessarily know *how* this is done. Good example of abstraction and encapsulation.

E.g. when you call "random.random()" in Python, you aren't concerned with how it works - just the output.

Lists

A list is a dynamic data structure that can contain an unlimited number of items which can be of different data types.

Methods:

```
isEmpty()           // Returns if the list is empty.
append(item)        // Adds a specified item to the end.
remove(item)        // Removes the first instance of the specified item.
search(item)        // Returns if the item is in the list or not.
length()            // Returns the length of the list.
index(item)         // Returns the index of the first occurrence of the item.
insert(pos, item)   // Inserts an item in a position, moving the already existing one after
it.
pop()               // Removes the last item in a list and returns it.
pop(pos)            // Removes an item by an index in a list and returns it.
```

Linked lists

A linked list is a dynamic data structure used to hold an ordered sequence

Queues

A first-in-first-out data structure (FIFO). New elements are added to the end, i.e. "enqueued", and items to be retrieved/removed are "dequeued".

Usage examples:

- Printer spooling.
- Task scheduling.
- Call centre systems.
- Handling website traffic.
- Characters typed on a keyboard.

Core methods:

```
isFull()         // Returns True or False depending on if the queue is full or not.
                 // True if the end pointer is equal to the length of the array.
                 // May not be needed if implemented with a list instead of an array.

isEmpty()        // Returns True or False depending on if the queue is empty or not.
                 // True if the end pointer is one less than the front pointer.
                 // (Not equal, because that indicates there is one item in the list)

enqueue(item)    // Adds an item to the end of the queue if it isn't full.

dequeue()        // Removes the first item in the queue if it isn't empty.
```

A queue is typically implemented with an array (or list), a front pointer, and an end pointer.

- The array ensures the queue only contains a fixed number of items of the same type.
- (A list may be used instead if the size of the queue cannot be determined beforehand.)
- The front pointer points to the first item in the queue. This is moved forward when "dequeue" is called.
- The end pointer points to the final item in a queue. This is moved forward when "enqueue" is called.

There are three types of queue:

Linear	• Can be implemented as a list or an array. • Pointers move forward. • As elements are dequeued, space at the front of the queue is never accessed again, so memory is wasted.
Circular	• Can be implemented as an array. • Pointers move forward, but circle back to the front of the array when the end is reached. • Fixes the issue of wasted memory. • Doesn't offer the flexibility of a dynamic data structure - best for queues in which the size can be predetermined. • More complicated for the programmer to implement.
Priority	• The logical order of items is determined by their priority, with higher priority items at the front. • It is possible for higher priority items to be enqueued to the front of the queue or inserted in between items.

Linked lists

Revision #5
Created 2026-02-25 11:32:36 UTC by Asad
Updated 2026-03-01 09:16:53 UTC by Asad