Topic 2 Fundamentals of Data Structures

Data Structures

Data structures can be thought of as boxes used by computers to store information. There are many different types of data structures, each with advantages and disadvantages and best suited to storing different types of data.

Arrays

An array is used to store related elements. They must contain a fixed number of elements, be indexed and can only contain elements with the same data type.

Each element within the array is given an index, usually starting at zero. The index is used to identify each element within the array, allowing them to be retrieved or edited.

Arrays can be either one dimensional or two dimensional. A one dimensional array is like a list, with one item after another. A two dimensional array is like a table, with values assigned two indexes depending on the column and row within the table.

Files, Records and Fields

Computers organise stored information into files, files contain one or more records, and records contain one or more fields.

Abstract Data Types and Structures

These don't exist as data structures in their own right and instead make use of existing structures to store data in a new way.

Queues

A queue is an abstract data type based on an array. Queues are referred to as first in first out, or FIFO because the first item added to a queue is the first to be removed.

Keyboard buffers queues to store keypresses. Each keypress is added to the queue, then removed when the keypress has been processed. Using a queue ensures the keypresses are processed in the same order in which they were typed.

These items can be performed on a queue:

- Enqueue – add an item to the queue.
- Dequeue – remove an item from the queue.
- IsEmpty – returns TRUE if the queue is empty.
- IsFull – returns TRUE if the queue is full.

Linear Queues

A linear queue uses a front point and a rear pointer to identify where new items should be placed and which item is at the front of a queue. The front pointer indicates the oldest item in the queue, and the rear pointer indicates where the next item should be added

Below is a queue with five positions, three are in use and two are free. The front pointer points to James, so we know that James is the oldest person in the queue.

James	Bob	Sally		

If the euqueue (add) operation was performed to enqueue Amanda, the rear pointer would move to the next free space and the queue would look like this.

James	Bob	Sally	Amanda	

If the dequeuer (remove) operation was performed, James as the oldest person in the queue would be removed and the queue would look like this. The front pointer now points at Bob as the oldest remaining person in the queue.

	Bob	Sally	Amanda	

Circular Queues

A circular queue allows the front and rear pointers to loop back to the start of the queue when the end is reached, making memory storage more efficient.

The example below shows a circular queue with five spaces, three free and to in use.

		Alex	Sarah	

If the enqeue operation is performed to enqueue Jenny, the rear pointer circles back to the next available space at the start of the queue. If this were a linear queue, the queue would now be full and no spaces available.

		Alex	Sarah	Jenny

Priority Queues

Priority queues assign a priority to items in the queue, higher priority items can be prioritised over lower priority items. If two or more items have the same priority they are processed in First In First Out order.

One example where priority queues are used is within the CPU. The CPU will prioritise resource requests from applications currently in use by the user to those running in the background.

Stacks

Stacks are a first in last out or FILO abstract data structure based on arrays. Stacks have a single top pointer which points to the item at the top of the stack.

The best way to think of stacks is as an actual stack of books. The first book in the stack will end up at the bottom and because of this be the last to be processed.

These operations can be performed on a stack:

- Push – add an item
- Pop – remove the item at the top
- Peek – return the value of the item of the top of the stack but do not remove it from the stack
- IsFull – returns TRUE if the stack is full
- IsEmpty – return TRUE if the stack is empty

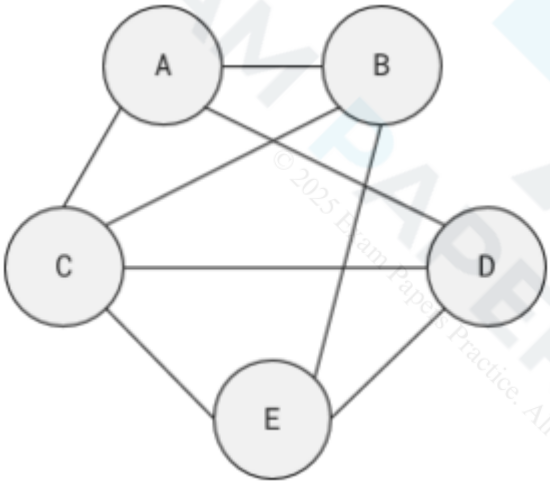
Graphs

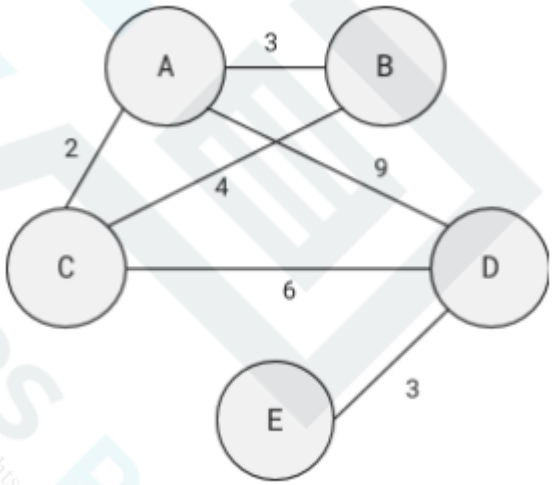
A graph is an abstract data structure which represents complex relationships. They are often used to represent computer networks.

A graph is made up of nodes which are joined together by edges. A weighted graph assigns values to edges to represent a distance or cost. A directed graph uses arrows as edges, allowing travel in one direction only.

Adjacency Matrices

An adjacency matrix is a table used to represent a graph with each node being assigned a row and column.





	A	B	C	D	E
A	0	1	1	1	0
B	1	0	1	0	1
C	1	1	0	1	1
D	1	0	1	0	1
E	0	1	1	1	0

	A	B	C	D	E
A	∞	3	2	9	∞
B	3	∞	4	∞	∞
C	2	4	∞	6	∞
D	9	∞	6	∞	3
E	∞	∞	∞	3	∞

A 1 shows that an edge exists between two nodes, 0 indicates that the two nodes do not have a direct connection. This gives a diagonal line down the middle of the graph where a node can't be connected to itself. The matrix displays diagonal symmetry, meaning the top right corner is the same as the bottom left.

When working with weighted graphs, the weight is used rather than a 1 or 0. Where a connection does not exist a fixed large value, usually infinity (∞) is used.

Adjacency Lists

Graphs can also be written as a list of adjacent nodes for each node rather than a matrix. The advantage here is that a list will only ever store connections which exist, whilst a matrix lists both those which do and do not exist.

Trees

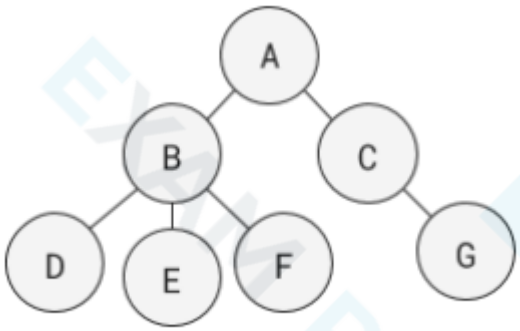
A Tree is a special kind of Graph which meets three criteria:

- Connected
- Undirected
- No cycles (meaning loops)

Rooted Trees

A rooted tree has a single root node, from which all other nodes stem. They are usually drawn with the root node at the top working down. The following terms describe the different nodes in a rooted tree:

- Parent nodes have one or more nodes stemming down from them
- Child nodes are connected to a parent nodes.
- Leaf nodes are child nodes which have no children connected to them.



Binary Trees

A binary tree is a rooted tree where each parent node has no more than two child nodes.

Hash Tables

Hash tables store data in a way that allows it to be retrieved with a constant time complexity of O(1). A hashing algorithm is used to convert an input into a hash and cannot convert a hash back to the input value. Hash algorithms must always return the same hash for a given input.

Hash tables store data in one column, and the hash (referred to as a key) in the other. To lookup a value in the table, it is first hashed and the hash used to lookup the vale in the table.

A hash algorithm must always produce the same hash for a given input, but sometimes two different input values can produce the same hash value. This is called a collision and hashing systems must be designed to work around collisions in order to avoid overwriting data.

The process of rehashing is used to work around collisions to use an agreed procedure to find a new position. One simple approach to rehashing is to keep moving to the next position until an available space is found.

Dictionaries

A dictionary is a collection of key value pairs in which a value is accessed by its associated key.

Vectors

Vectors can be shown in four ways:

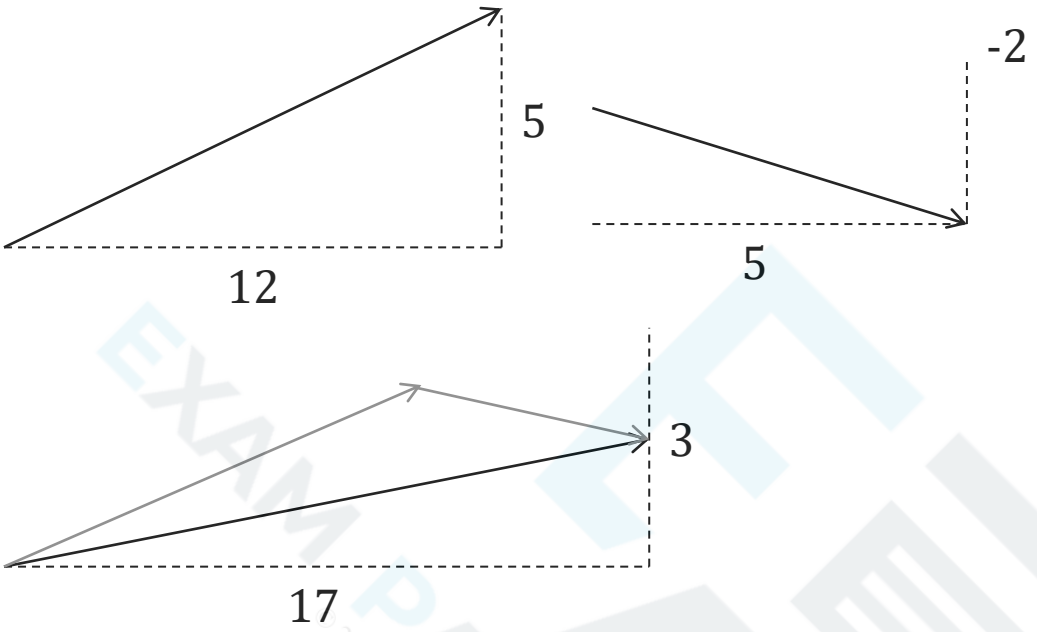
- As a list of numbers – [22, 4, 9, 26]
- As a vector space over a field – 4-vector over $\mathbb{Z}$ ($\mathbb{Z}^4$)
- As a function - 0 ↦ 12

- As a point in space (22,9,4,26)

If show as a function a vector could be represented using a dictionary, whilst if viewed as a list a one dimensional array would be more suitable. A vector can also be drawn arrow, making them easier to visualise.

Vector Addition

Translation can be achieved by adding vectors together, this must always be done tip to tale:



This could also be written like this:

[12, 5]
+ [5, -2]
= [17, 3]

Scalar Vector Multiplication

To scale a vector, each component must be multiplied by the same scalar as shown below. This process affects the size (magnitude) of the vector but not its direction.

[12, 2]
X 2
= [24, 4]

Convex Combination of Two Vectors

A convex combination of two vectors is formed on a line which would join the tips of the vectors if they were both drawn as arrows. The convex combination splits this line in the ratio chosen for x and y.

A convex combination of two vectors can be calculated using the formula ax + by. Both x and y must be non-zero, less than one and add up to one.

Example

```
vector a = [1, 1] vector b = [2, 4]
x = 0.8 and y = 0.2
ax (a multiplied by x) = [0.8, 0.8]
by (b multiplied by y) = [0.4, 0.8]
ax + by = [1.2, 1.6]
```

This example calculates the vector needed to split the line in a 0.2:0.8 ratio, or 20% of the say from a to b.

Dot Product

The dot product (also known as scalar product) of two vectors is a single number which is derived from the components of the vectors and which can be used to work out the angle between the two vectors.

This is calculated by multiplying the x parts of the two vectors, the y parts of the two vectors and adding the result together as shown below.

Example

```
vector a = [12, 3] vector b = [5, 8]
a • b = (12 x 5) + (3 x 8)
a • b = 84
```

Static and Dynamic Data Structures

Data structures can be dynamic or static. A dynamic data structure can change in size as needed to store content, whilst a fixed data structure will always remain one fixed size.

Static data structures are usually declared in memory as a series of sequential contiguous memory locations, making them easier for the computer to maintain. If the number of data items in a fixed structure exceeds the number of available memory locations allocated an overflow error occurs.

Dynamic structures are more complicated for the computer to maintain. Since the number of memory locations needed changes, the computer can't allocate contiguous memory locations and instead each item has to be stored alongside a reference explaining where the next item is stored. This allows dynamic structures to be as big or small as needed to, but requires more work to setup and use.

If the number of data items to be stored in a dynamic data structure exceeds the number of memory locations allocated, new memory locations are simply added until there is enough space for the data.