AQA Computer Science A-Level

4.2.1 Data structures and abstract data types

Advanced Notes

Specification:

4.2.1.1 Data structures:

Be familiar with the concept of data structures.

4.2.1.2 Single- and multi-dimensional arrays (or equivalent):

Use arrays (or equivalent) in the design of solutions to simple problems.

4.2.1.3 Fields, records and files:

Be able to read/write from/to a text file.

Be able to read/write data from/to a binary (non-text) file.

4.2.1.4 Abstract data types/data structures:

Be familiar with the concept and uses of a:

- Queue
- Stack
- Graph
- Tree
- Hash table
- Dictionary
- Vector

Be able to distinguish between static and dynamic structures and compare their uses, as well as explaining the advantages and disadvantages of each.

Describe the creation and maintenance of data within:

- Queues (linear, circular, priority)
- Stacks
- Hash tables

4.2.2.1 Queues:

Be able to describe and apply the following to linear queues, circular queues and priority queues:

- add an item
- remove an item
- test for an empty queue
- test for a full queue

4.2.3.1 Stacks:

Be able to describe and apply the following operations:

- push
- pop
- peek or top
- test for empty stack
- test for stack full

4.2.4.1 Graphs:

Be aware of a graph as a data structure used to represent more complex relationships.

Be familiar with typical uses for graphs.

Be able to explain the terms:

- graph
- weighted graph
- vertex/node
- edge/arc
- undirected graph
- directed graph.

Know how an adjacency matrix and an adjacency list may be used to represent a graph.

Be able to compare the use of adjacency matrices and adjacency lists.

4.2.5.1 Trees (including binary trees):

Know that a tree is a connected, undirected graph with no cycles.

Know that a rooted tree is a tree in which one vertex has been designated as the root. A rooted tree has parent-child relationships between nodes. The root is the only node with no parent and all other nodes are descendants of the root.

Know that a binary tree is a rooted tree in which each node has at most two children.

Be familiar with typical uses for rooted trees.

4.2.6.1 Hash tables:

Be familiar with the concept of a hash table and its uses.

Be able to apply simple hashing algorithms.

Know what is meant by a collision and how collisions are handled using rehashing.

4.2.7.1 Dictionaries:

Be familiar with the concept of a dictionary

Be familiar with simple applications of dictionaries, for example information retrieval, and have experience of using a dictionary data structure in a programming language.

4.2.8.1 Vectors:

Be familiar with the concept of a vector and the following notations for specifying a vector:

- [2.0, 3.14159, -1.0, 2.718281828]
- 4-vector over $\mathbb{R}$ written as $\mathbb{R}4$
- function interpretation
- $0 \mapsto 2.0$
- $1 \mapsto 3.14159$
- $2 \mapsto -1.0$
- $3 \mapsto 2.718281828$
- $\mapsto$ means maps to

That all the entries must be drawn from the same field, eg $\mathbb{R}$.

Dictionary representation of a vector.

List representation of a vector.

1-D array representation of a vector.

Visualising a vector as an arrow.

Vector addition and scalar-vector multiplication.

Convex combination of two vectors, u and v.

Dot or scalar product of two vectors.

Applications of dot product.

Data structures

Data structures are used by computers as the containers within which information is stored. Different data structures exist and some are better suited to different types of data than others. When storing data, a programmer must decide which of the data structures available is the best to use.

Arrays

An array is an indexed set of related elements. An array must be finite, indexed and must only contain elements with the same data type.

```
Array Names = {"George", "Sue", "Mo"}
```

The elements of an array are given an index, which often starts from zero. For example, with the array shown above, `Names(2)` would return `"Mo"` as the first item (`"George"`) is given the index `0`.

Synoptic Link

A **data type** is defined by the values it can take or the operations which can be performed on it.

Data types are covered in the notes for **fundamentals of programming**.

The array shown above is a one-dimensional array which could be visualised with the following table:

0	1	2
"George"	"Sue"	"Mo"

Arrays can be created in many dimensions. For example, a two-dimensional array could look like this:

```
Array Maze = { {Wall, Path, Wall}, {Path, Path, Wall}, {Wall, Path, Wall}}
```

When displayed in a table, the `Maze` array starts to make a little more sense:

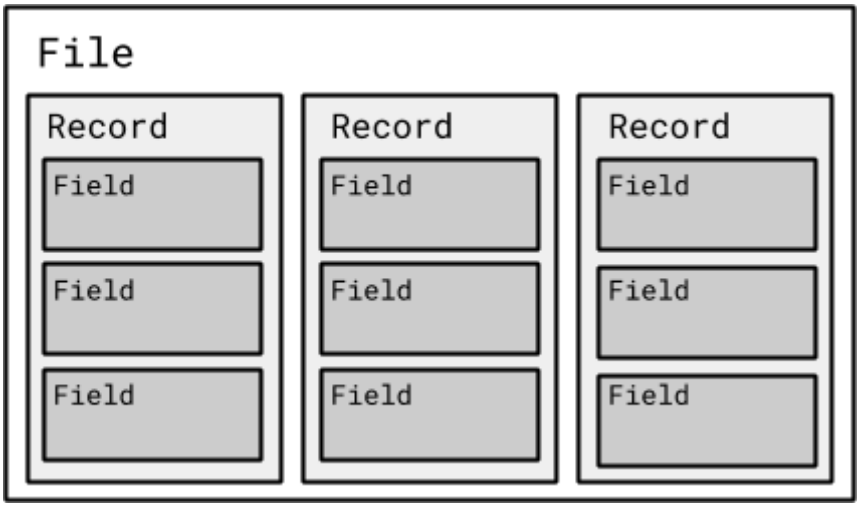
	0	1	2
0	Wall	Path	Wall
1	Path	Path	Wall
2	Wall	Path	Wall

When an individual element is referenced, the x coordinate is listed first.

For example, `Maze(1,2)` would return `Path` and `Maze(2,1)` would return `Wall`.

Fields, records and files

Information is stored by computers as a series of files. Each file is made up of records which are composed of a number of fields.



It's important that you make sure you can write to and read from files in your chosen programming language.

Abstract data types/data structures

Abstract data structures don't exist as data structures in their own right, instead they make use of other data structures such as arrays to form a new way of storing data.

Queues

A queue is an abstract data structure based on an array. Just like a queue at a bus stop, the first item added to a queue is the first to be removed. Because of this, queues are referred to as "first in, first out" (or FIFO) abstract data structures.

Queues are used by computers in keyboard buffers, where each keypress is added to the queue and then removed when the computer has processed the keypress. This ensures that letters appear on the screen in the same order that they were typed.

The breadth first search algorithm uses a queue to keep track of which nodes in a network have been visited.

Synoptic Link

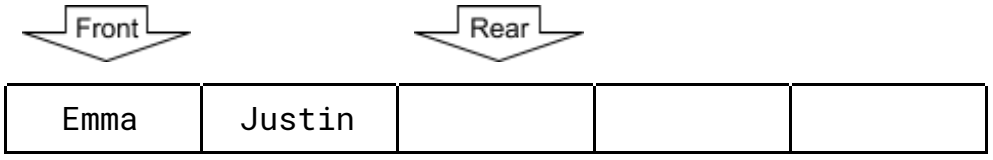
The **breadth first search algorithm** is used for finding an item in a network.

Breadth first search is covered in **graph traversal** under **fundamentals of algorithms**.

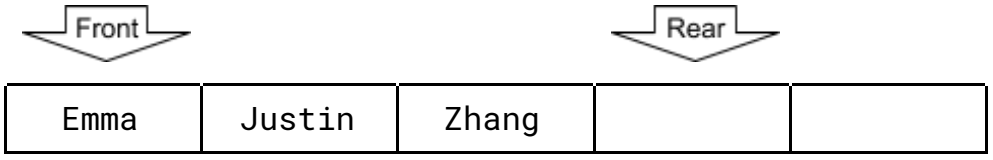
Linear queues

A linear queue has two pointers, a front pointer and a rear pointer. These can be used to identify where to place a new item in a queue or to identify which item is at the front of the queue.

The example below shows a queue with five positions, two of which are occupied. Emma is at the front of the queue so must have been enqueued (added to the queue) before Justin, who is at the back of the queue. The rear pointer always points to the next available position in the queue.

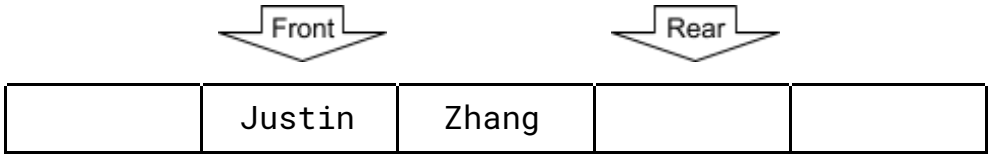


If we were to perform the operation `Queue.Enqueue("Zhang")`, the queue would look like this:



Zhang is added behind Justin and the rear pointer moves to the next available position.

If we were to perform the operation `Queue.Dequeue()`, the queue would look like this:



Emma, who was at the position of the front pointer, has been dequeued and the front pointer has moved to Justin.

If the dequeue operation were to be performed again, Justin would be removed from the queue, as he is now at the front of the queue.

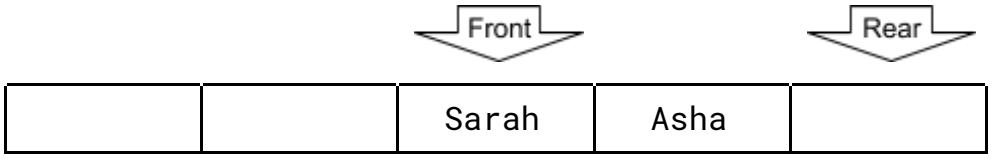
Note

There's no need to specify a person in a dequeue command.

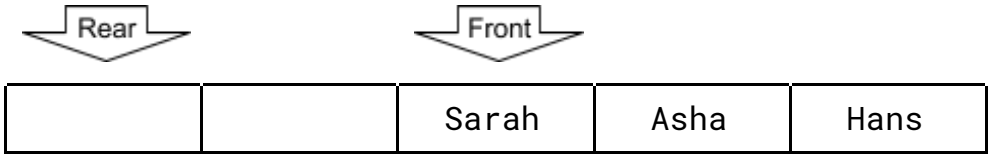
Operations that can be performed on a queue include `Enqueue`, `Dequeue`, `IsEmpty` and `IsFull`. `IsEmpty` is a function that returns TRUE if the queue has no content. Emptiness can be detected by comparing the front and rear pointers. If they are the same, the queue is empty. `IsFull` is a function that returns TRUE if the queue has no available positions that are behind the front pointer.

Circular queues

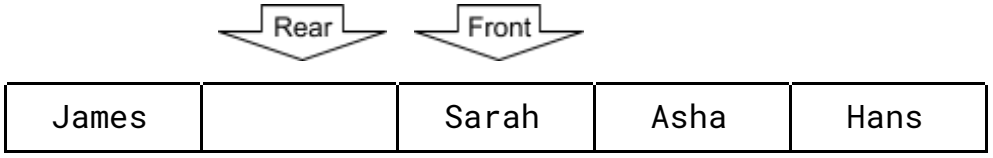
A circular queue is a type of queue in which the front and rear pointers can move over the two ends of the queue, making for a more memory efficient data structure.



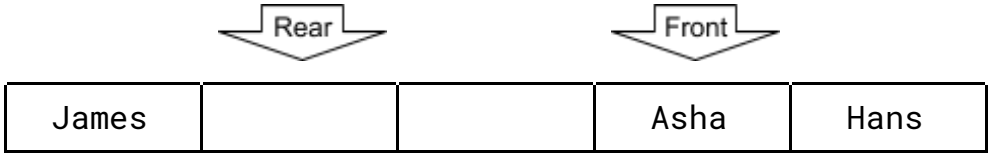
When the operation `Queue.Enqueue("Hans")` is performed on the circular queue above, the rear pointer jumps to the left hand side of the queue.



Now the operation `Queue.Enqueue("James")` can be performed. This would not have been possible if the queue had been implemented as a linear queue as there were no available spaces behind the front pointer. In a circular queue however, the rear pointer can jump over the ends of the queue and make use of available spaces before the front pointer.



The queue continues to work as before, dequeuing items from the position specified by the front pointer. If the operation `Queue.Dequeue()` were performed now, Sarah would be removed, placing Asha at the front of the queue.



Priority Queues

In a priority queue, items are assigned a priority. Items with the highest priority are dequeued before low priority items.

In the case that two or more items have the same priority, the items are removed in the usual *first in, first out* order.

Priority queues are frequently used in computer systems. For example, processors assign time to applications according to their priority. Applications currently in use by the user are prioritised over background applications, allowing for a faster user experience.

A school or college may enforce a priority printer queue, in which staff print jobs are completed before those submitted by students.

Stacks

A stack is a first in, last out (FILO) abstract data structure. Like queues, stacks are often based on an array but have just one pointer: a top pointer.

You can think of stacks as a pile of students' books which need to be marked by a teacher. The first book to be handed in is placed at the bottom of the stack and is the last to be marked. The last book to be added to the stack is the first to be marked.

Operations that can be performed on a stack include `Push` (add an item), `Pop` (remove the item at the top) and `Peek`. `Peek` is a function which returns the value of the item at the top of the stack without actually removing the item. The functions `IsFull` and `IsEmpty` can also be executed on stacks, just like with a queue.

The stack on the left contains three names. Bruce is at the top of the stack, as indicated by the position of the top pointer.

Mary is at the bottom of the stack, which must mean that Mary has been in the stack for the longest.

The operation `Stack.Peek()` would return "`Bruce`" but would not change the stack in any way.

The diagram on the left shows the state of the stack after the operation `Stack.Push("Charlie")`.

Executing the operation `Stack.Pop` at this stage would remove Charlie from the stack and move the top pointer down one position to Bruce.

The pop operation can be used to assign variables and constants. For example, the operation `x ← Stack.Pop` would assign the value "`Charlie`" to the variable `x`. This is also applicable to the peek function.

The diagram on the left shows the state of the stack after the operation `Stack.Push("Maria")`.

Maria is now at the top of the stack and there are no available spaces above the top pointer.

If a push command were executed now, an error would be returned. This error is called a stack overflow and indicates that there are no available spaces on the stack.

A similar error, called a stack underflow, can be caused by attempting the pop command on an empty stack.

Graphs

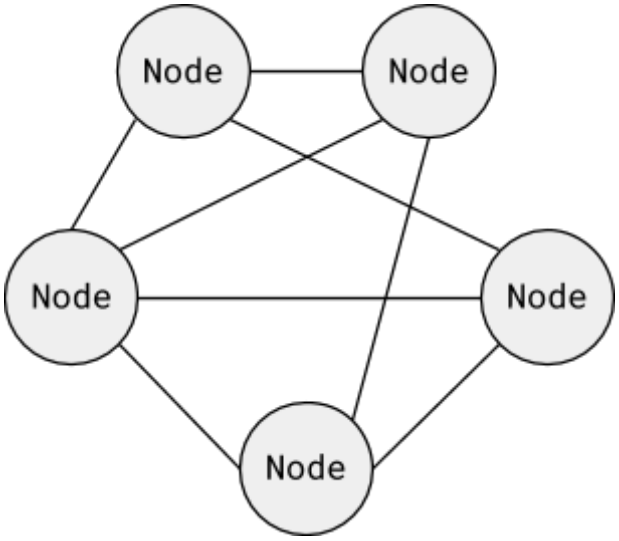
A graph is an abstract data structure used to represent complex relationships between items within datasets. Graphs can be used to represent networks such as transport networks, IT networks and the Internet.

A graph consists of nodes (sometimes called vertices) which are joined by edges (sometimes called arcs). A weighted graph is one in which edges are assigned a value, representing a value such as time, distance or cost.

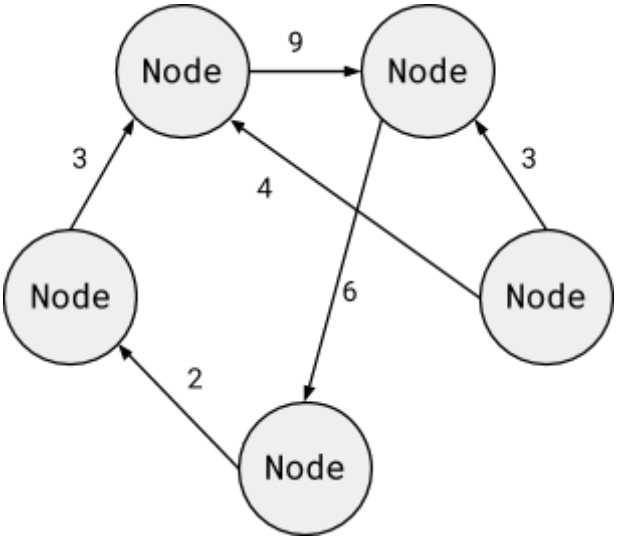
Synoptic Link

The Internet is a network of interconnected computer networks.

This is covered in more detail in **the structure of the internet** under **fundamentals of communication and networking**.



Unweighted, undirected graph



Weighted, directed graph

Graphs can be represented in two different ways. Using adjacency matrices or adjacency lists. Each of the two methods has its own relative advantages and disadvantages, so choosing the right method of representing a graph is essential when developing a solution.

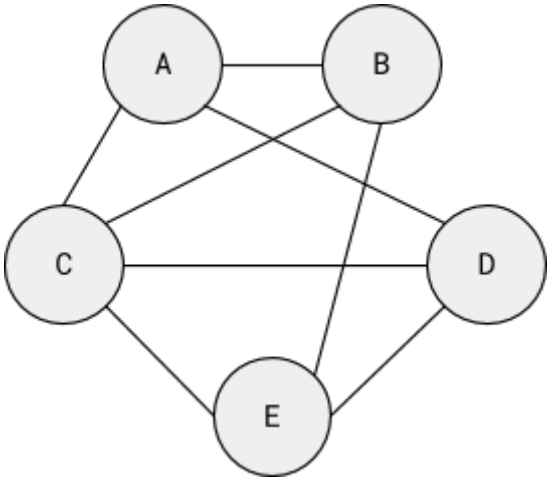
Note

Weighted graphs are rarely drawn to scale.

Adjacency matrices

An adjacency matrix is a tabular representation of a graph. Each of the nodes in the graph is assigned both a row and a column in the table.

Example: Unweighted graph



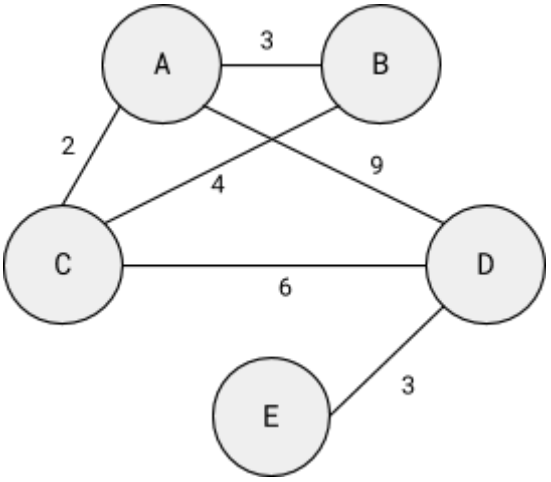
	A	B	C	D	E
A	0	1	1	1	0
B	1	0	1	0	1
C	1	1	0	1	1
D	1	0	1	0	1
E	0	1	1	1	0

A 1 is used to show that an edge exists between two nodes and 0 indicates that there is no connection.

Notice that adjacency matrices have a characteristic diagonal line of 0s (shown in blue) where the row and column represent the same node.

Note also that adjacency matrices display diagonal symmetry, as shown by the green cells which are a reflection of the white cells in the blue diagonal line.

Example: Weighted graph



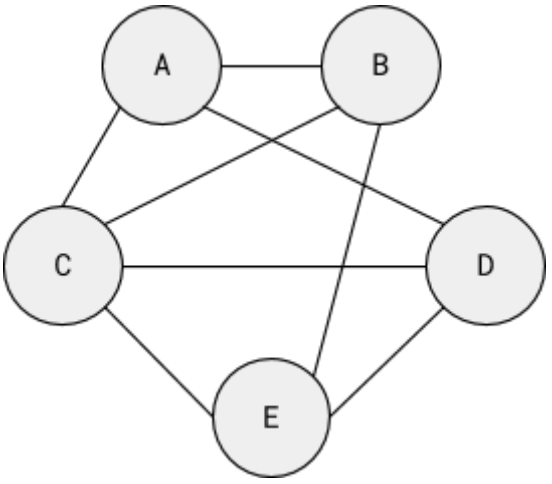
	A	B	C	D	E
A	∞	3	2	9	∞
B	3	∞	4	∞	∞
C	2	4	∞	6	∞
D	9	∞	6	∞	3
E	∞	∞	∞	3	∞

Rather than using Boolean values, adjacency matrices for weighted graphs contain the weight of a connection between two nodes.

If no connection exists, an arbitrarily large value is used. This will prevent any shortest path algorithm from using any non-existent edges. When written down, this value is often expressed as infinity.

Adjacency lists

Rather than using a table to represent a graph, a list could be used.



A B, C, D
B A, C, E
C A, B, D, E
D A, C, E
E B, C, D

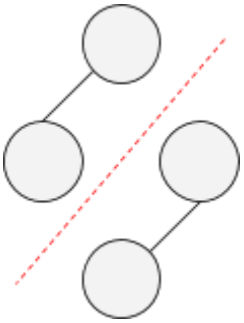
The graph on the left could be represented with the adjacency list on the right.

For each node in the graph, a list of adjacent nodes is created. This means that an adjacency list only records existing connections in a graph, unlike an adjacency matrix which stores even those edges that don't exist.

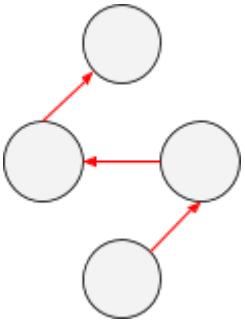
Adjacency matrix	Adjacency list
Stores every possible edge between nodes, even those that don't exist. Almost half of the matrix is repeated data. **Memory inefficient.**	Only stores the edges that exist in the graph. **Memory efficient.**
Allows a specific edge to be queried very quickly, as it can be looked up by its row and column. **Time efficient.**	Slow to query, as each item in a list must be searched sequentially until the desired edge is found. **Time inefficient.**
Well suited to dense graphs, where there are a large number of edges.	Well suited to sparse graphs, where there are few edges.

Trees

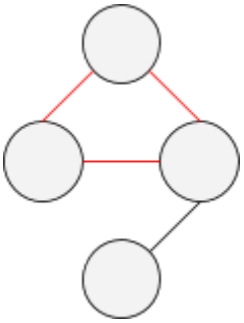
A tree is a connected, undirected graph with no cycles.



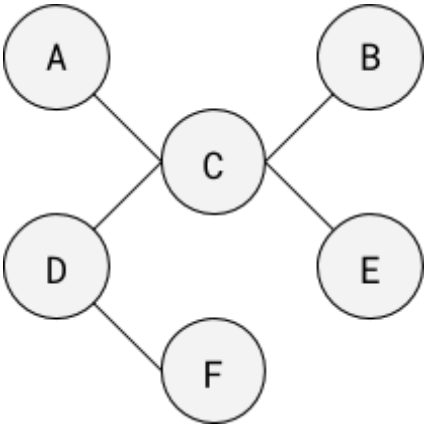
✘ Connected
✔ Undirected
✔ No cycles



✔ Connected
✘ Undirected
✔ No cycles



✔ Connected
✔ Undirected
✘ No cycles



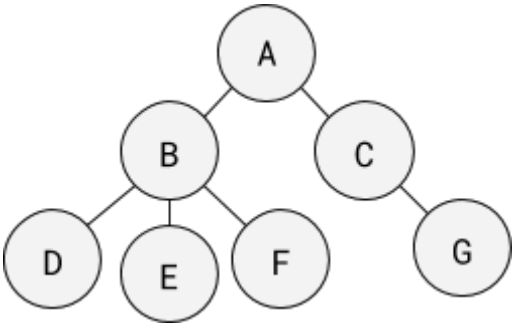
None of the three graphs above are trees. The first is not connected, the second is directed and the third contains a cycle.

The graph on the left is a tree. All of the nodes are connected, the edges are undirected and there are no cycles.

Rooted trees

A rooted tree has a root node from which all other nodes stem. When displayed as a diagram, the root node is usually at the top of the tree.

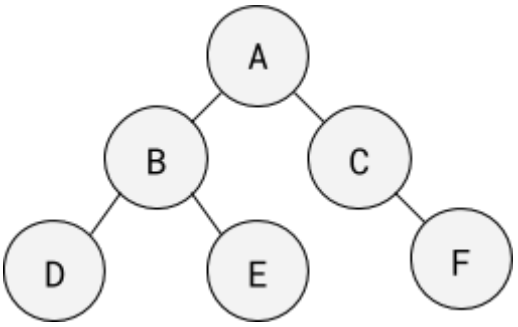
Nodes from which other nodes stem are called parent nodes. The root node is the only node in a rooted tree with no parent. Nodes with a parent are called child nodes and child nodes with no children are called leaf nodes.



Root: A
Parent: A, B, C
Child: B, C, D, E, F, G
Leaf: D, E, F, G

Binary trees

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



Synoptic Link

A common application of a **binary tree** is as a **binary search tree**.

Binary search trees are covered in **binary tree search** under **fundamentals of algorithms**.

Rooted trees (which include Binary trees) can be used to represent family trees or file systems on a computer's hard drive.

Hash tables

Hash tables are a way of storing data that allows data to be retrieved with a constant time complexity of O(1).

A hashing algorithm takes an input and returns a hash. A hash is unique to its input and cannot be reversed to retrieve the input value.

For example, a simple hashing algorithm is:

```
Value ← INPUT
Hash  ← Value MOD 3
RETURN Hash
```

If the input were 10, the hash would be 1. The same hash is always returned for each input.

Synoptic Link

O(1) is an example of **big O notation**.

Big O notation is covered in **classification of algorithms** under **theory of computation**.

Note

Passwords are **hashed** before being stored so that your password cannot be retrieved even if the hash is exposed.

A hash table stores key-value pairs. The key is sent to a hash function that performs arithmetic operations on it. The resulting hash is the index of the key-value pair in the hash table.

When an element is to be looked up in a hash table, the key is first hashed. Once the hash has been calculated, the position in the table corresponding to that hash is queried and the desired information is located.

Sometimes different inputs produce the same hash, for example: the hashing algorithm above produces the same hash for the values 12 and 30. This is called a collision, and could result in data being overwritten in a hash table in a poorly designed system.

Well designed hash tables get around collisions by using rehashing, finding an available position according to an agreed procedure.

One simple rehashing technique is to keep on moving to the next position until an available one is found.

The hash table below stores buildings using a hash of their height as the key. The hashing algorithm being used is the one shown previously, which takes the building's height and applies the `modulo 3` operation to it.

When "The Shard (310m)" is added to the hash table, the key is calculated by the hashing algorithm as 1 (`310 MOD 3 = 1`). However, position 1 is already occupied by The Great Pyramid of Giza. A collision has occurred.

Key	Value
0	
1	Great Pyramid of Giza (139m)
2	Empire State Building (443m)
3	
4	

In order to get around this collision, a rehashing algorithm is carried out. The next position (position 2) is inspected, but is occupied, so the algorithm moves on to position 3. Position 3 is unoccupied and so The Shard is placed in position 3.

Key	Value
0	
1	Great Pyramid of Giza (139m)
2	Empire State Building (443m)
3	The Shard (310m)
4	

When The Shard is retrieved from the hash table, the hash is calculated as 1, so the value in position 1 is queried. The value in position 1 is not the requested value, and so the next position is queried. This continues until the desired value is found in position 3.

Dictionaries

A dictionary is a collection of key-value pairs in which the value is accessed by its associated key. For example, the words in the sentence “row, row, row your boat” could be put into a dictionary like the one below:

> **Synoptic Link**
>
> **Dictionary-based compression** is covered in more detail under **fundamentals of data representation.**

Key	Value
1	row
2	your
3	boat

One application of dictionaries is in dictionary-based compression. The sentence “row, row, row your boat” could be compressed as 11123 using the dictionary above.

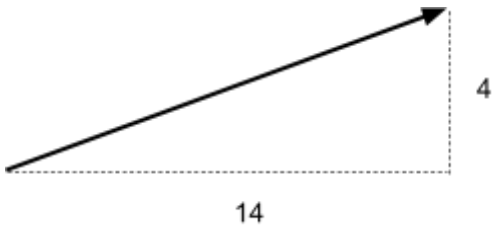
Vectors

Vectors can be represented lists of numbers, functions or ways of representing a point in space.

As a list of numbers	`[12, 7, 3, 55]`
As a vector space over a field	4-vector over $\mathbb{Z}$ ($\mathbb{Z}^4$)
As a function	$0 \mapsto 12$ $1 \mapsto 7$ $2 \mapsto 3$ $3 \mapsto 55$
As a point in space	(12, 7, 3, 55)

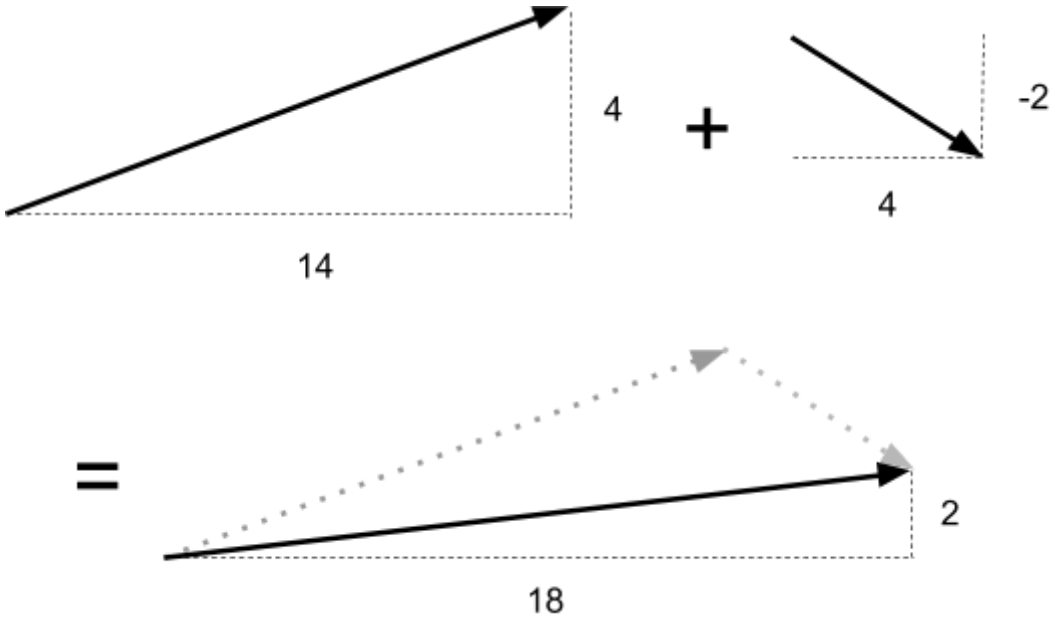
If viewed as a function, a vector can be represented by using a dictionary. If viewed as a list, a one-dimensional array would be suitable.

A vector can be visually represented as an arrow, for example: the two dimensional vector [14, 4] could be represented with the arrow below.



Vector addition

Vectors can be added to achieve translation. With arrows, vectors are added “tip to tail” as in the example below:



Alternatively, vectors can be added by adding each of their components like so:

$$\begin{aligned} & [14,\ 4] \\ + & [4,\ -2] \\ = & [18,\ 2] \end{aligned}$$

Scalar-vector multiplication

In order to scale a vector, each of its components are multiplied by a scalar.

$$\begin{aligned} & [14,\ 4] \\ \times & \quad 3 \\ = & [42,\ 12] \end{aligned}$$

Scaling a vector affects only its magnitude, not its direction.

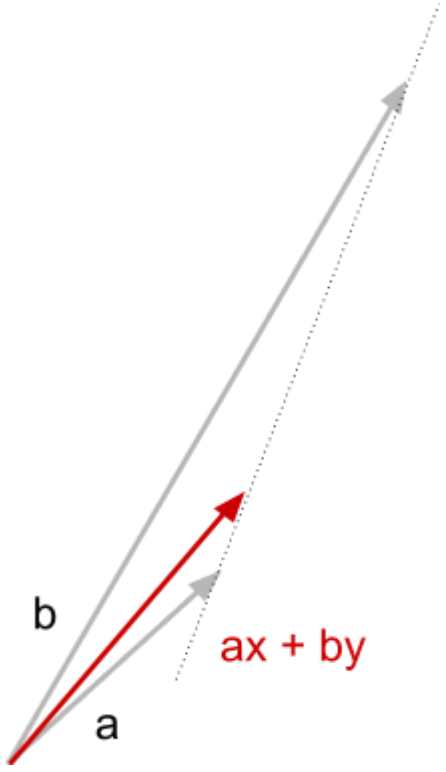
Convex combination of two vectors

If you have two vectors, a and b, then a convex combination of the two would be $ax + by$ where x and y are both non-zero numbers less than one that add to 1.

For example, let a be the vector [1, 1] and b be the vector [2, 4].

Let's choose x and y to be 0.8 and 0.2 respectively. Neither are zero and they add to 1.

Now we can form a convex combination of a and b as follows:



$$ax = [0.8, 0.8]$$
$$by = [0.4, 0.8]$$
$$ax + by = [1.2, 1.6]$$

As the diagram on the left shows, the convex combination of a and b is formed on the line that would join the tips of a and b.

The convex combination splits the line joining the tips of a and b in the ratio chosen for the values x and y.

In this case, the line has been split between 0.2 and 0.8 (20% of the way from the tip of a to the tip of b).

Dot product

The dot product of two vectors (also called the scalar product) is a single number derived from the components of the vectors that can be used to determine the angle between two vectors.

The dot product of the vectors $a = [12, 3]$ and $b = [5, 8]$ is notated $a \cdot b$ (said "a dot b") and is calculated as follows:

$$a \cdot b = (12 \times 5) + (3 \times 8)$$
$$= 84$$

Static and dynamic data structures

Every data structure is either static or dynamic. Static data structures are fixed in size whereas dynamic data structures change in size to store their content.

Static data structures are most frequently declared in memory as a series of sequential, contiguous memory locations which makes them simply for the computer to maintain. The next element will simply be in the memory location next door.

If the number of data items to be stored in a static structure exceeds the number of memory locations allocated to the structure, an overflow error (such as a stack overflow) will occur.

Dynamic data structures are more complicated. Because the number of memory locations required isn't fixed, the computer simply can't allocate them contiguous memory locations. Instead, each item of data in the structure is stored alongside a reference to where the next item is stored in memory. This allows dynamic data structures to be as big or as small as they need to be but requires more work on the part of the computer to set up and use.

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