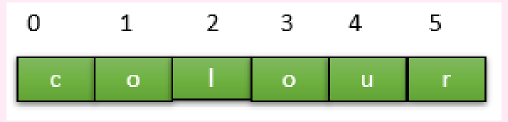
Abstract Data Types

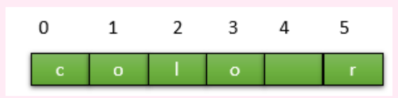
Abstract data types allow us deal with the operations and behaviours of a data type and not to be concerned with their operation which is abstracted away.

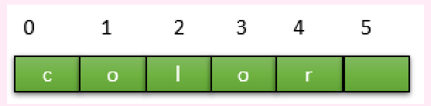
Data Structures

Static data structure
This is a fixed block of memory that is reserved at the start of the program. This is a contiguous space on disk. The next memory location is the next address and its position can be implied, so there is no need to explicitly point to it.

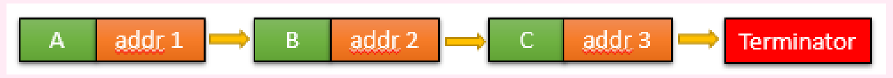


Suppose we want to remove the 'u'. This is not easy for static memory location because we must move all the succeeding elements up one place.

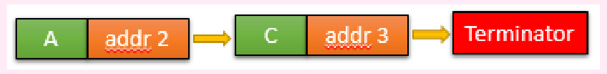




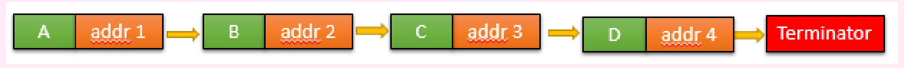
Dynamic data structure
Dynamic memory allocation is where memory is allocated and deallocated during the running of the program. The memory is allocated on the heap. The heap allows random allocation and access of memory. Dynamic memory allocation uses linked lists where each element points to the address of the succeeding element.



To remove an element just requires pointing to a different address



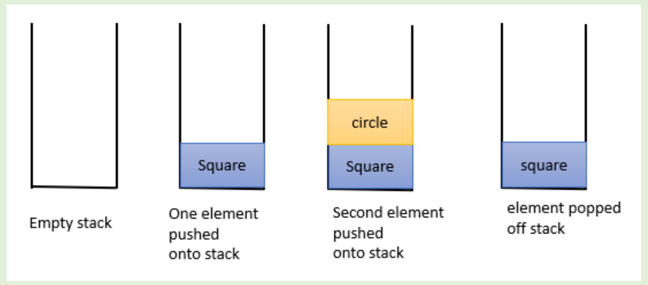
Conversely to add an element just requires pointing to that address



	Advantages	Disadvantages
Static data structures	Memory locations are fixed and can be accessed easily and quickly and are in a contiguous position in memory	Memory is allocated even when it is not being used.
Dynamic data structures	More flexible and more efficient than static data structures because we only use memory that is needed. Uses linked lists and makes it much easier to remove and add element.	Data structure may be fragmented so can be slow to access.

Stacks

Stacks are a *last in first out* file system just like a stack of plates. That is the last item added to the stack is the first to be retrieved.


Empty stack
Square
One element pushed onto stack
circle
Square
Second element pushed onto stack
square
element popped off stack

Stack operations
push – add element to the stack
pop – remove element from the stack
peek / top – view the top element on a stack without removing
isEmpty – test to see if stack is empty
isFull – test to see if stack is Full

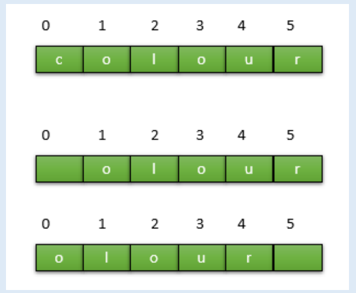
Uses of stacks
- Can reverse a sequence of numbers by popping a value from one stack and pushing to another
- Used in Reverse Polish Notation.
- Stack frames used in subroutine calls

Queues

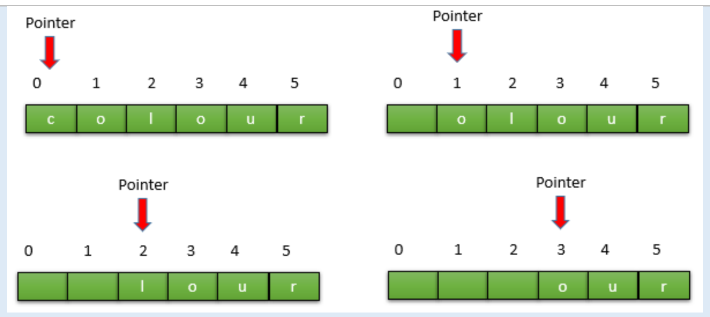
A queue is a **first in first out** data structure. Typically queues are used in buffering where a sequence of instructions are sent to a printer for instance, and the printer prints of the documents in the order in which the instructions arrived. Lists can be used to represent queues

Queue operations
- *Add* – add element to the end of a queues
- *remove* – remove element from front of queue
- *isEmpty* – test to see if queue is empty
- *isFull* – test to see if queue is Full

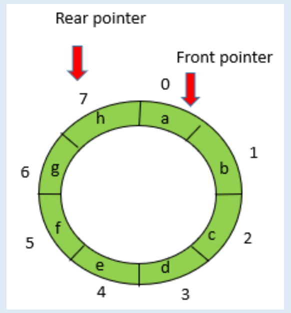
Linear queue
As an item is removed from the queue all the other items move up one space. For a long queue this can take a lot of processing.



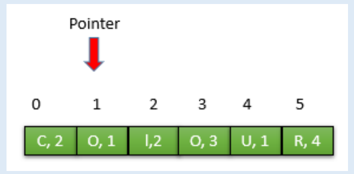
Linear queue using pointers
As an item is removed from the queue the pointer representing the start of the queue also moves up one. We need to know the length of the queue and how many elements have been removed. The problem with this method is that we end up with a lot of empty cells in memory that are now unused at the front of the list.



Circular queue
In a linear array when item are removed from the memory location those memory locations are allocated but are no longer used. Circular queues get around this problem by "recycling" theses memory locations. So that the memory locations at the front of the queue now become memory locations at the back of the queue



Priority Queue
Each element is assigned a priority. Highest priority items are removed first. If elements have the same priority then the item nearest the front of the queue is removed first. So in this case O would be the first item removed.



Alternatively, the queue could store items in priority order and the item removed from the front of the queue as with a linear queue

Graphs

A graph is a way of representing the relation between data. A graph is made up of vertices/nodes that are connected by edges or arcs. This could represent a rail or road network.

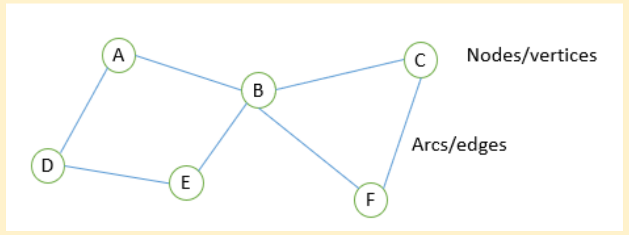
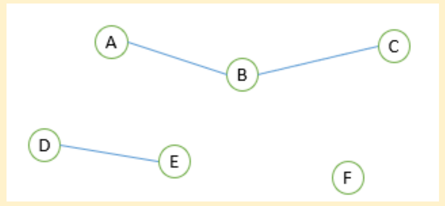


Figure 1

Graphs do not need to be connected. This is a valid graph.



Weighted graph
Weighted graphs add a value to an arc. This might represent the distance between places or the time taken between train stations

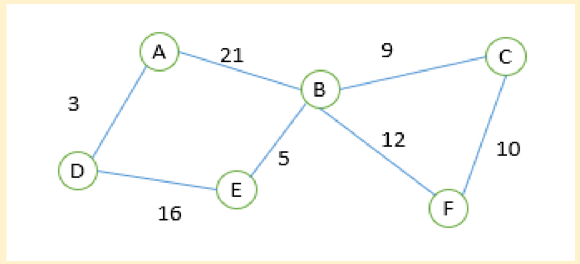


Figure 2

Adjacency Matrix With No Weighting

- Graphs can be represented as adjacency matrices.
- Graphs with no weights are given a value of 1 for connected nodes.

	A	**B**	**C**	**D**	**E**	**F**
A	-	1	-	1	-	-
B	1	-	1	-	1	1
C	-	1	-	-	-	1
D	1	-	-	-	1	-
E	-	1	-	1	-	-
F	-	1	1	-	-	-

Adjacency Matrix With Weighting

	A	B	C	D	E	F
A	-	21	-	3	-	-
B	21	-	9	-	5	12
C	-	9	-	-	-	10
D	3	-	-	-	16	-
E	-	5	-	16	-	-
F	-	12	10	-	-	-

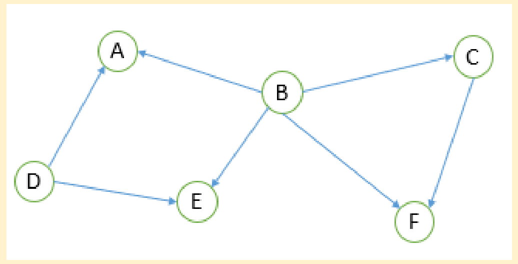
Adjacency List With No Weighting
Graphs can also be represented as adjacency lists. Adjacency list for Figure 1.

A	[D, B]
B	[A, E, C,F]
C	[B, F]
D	[A, E]
E	[D, B]
F	[B, C]

Adjacency List With Weighting
Graphs can also be represented as adjacency lists. Adjacency list for Figure 2.

A	{D:3, B:21}
B	{A:21, E:5, C:9, F:12}
C	{B:9, F:10}
D	{A:3, E:16}
E	{D:16, B:5}
F	{B:12, C:10}

Directed Graphs
Undirected graphs have connections in both directions. Directed graphs only apply in one direction and are represented with edges with arrow heads on one end.



Directed graph as adjacency list

A	
B	[A, E, C, F]
C	[F]
D	[A, E]
E	
F	

Directed graph as adjacency matrix

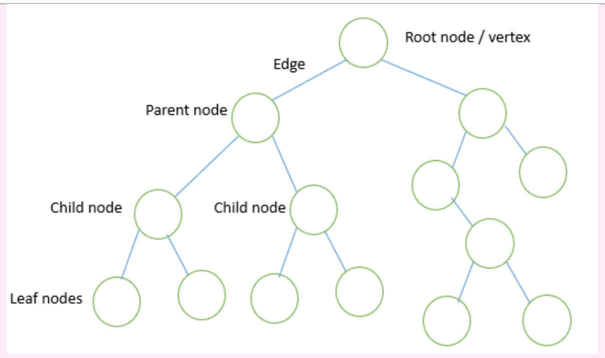
		To					
		A	**B**	**C**	**D**	**E**	**F**
From	**A**						
	B	1		1		1	1
	C						1
	D	1				1	
	E						
	F						

Adjacency matrices versus Adjacency lists

- For sparse graphs where there are a few edges there will be a lot of empty cells in the adjacency matrices thereby taking up more unused computer memory
- Adjacency lists take longer to process so are slower
- Deciding whether to use adjacency matrices or lists depends on the application
- For sparse graphs where memory is a limiting factor adjacency lists are preferable
- For graphs with lots of edge adjacency matrix is preferable

Trees

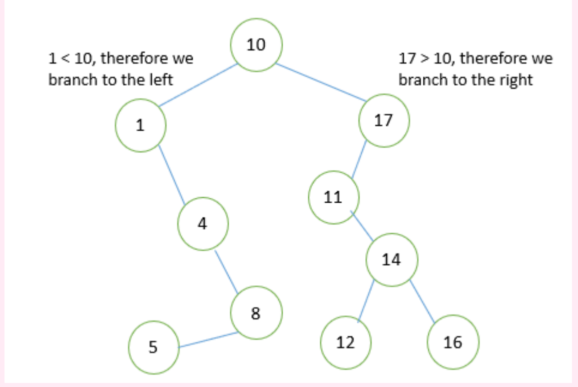
- A tree is a connected, undirected graph with no cycles.
- *Connected* - Every node is connected either indirectly to directly to every other node.
- *No Cycles* – There is only one path between nodes
- *Undirected* - can traverse in both directions along the edge
- A *rooted tree* has a root node that has no parent and all other nodes are descended from the root. All other nodes can be a parent and/or a child node.
- A *leaf node* has no children



Binary Tree

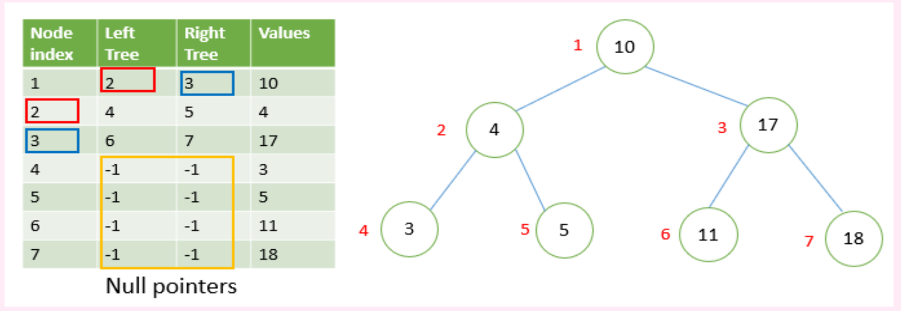
- In a binary tree a node can only have a maximum of two child nodes
- A binary tree can be used for sorting a sequence of numbers
- The first number is the root node
- If the number is smaller than the node then we branch left if it is bigger we branch right.

A binary tree for a sequence of numbers: 10,1,17,4,8,11,14,16,5,12


1 < 10, therefore we branch to the left
17 > 10, therefore we branch to the right
10
1
17
4
11
14
8
5
12
16

Tree data structure

- We can represent a tree data structure with three lists/arrays.
- An array contains the values at the nodes
- An array that points to the location of left child of the node in the values array
- An array that points to the location of right child of the node in the values array
- If a node does not have child node then this is indicated with a -1 or null



Node index	Left Tree	Right Tree	Values
1	2	3	10
2	4	5	4
3	6	7	17
4	-1	-1	3
5	-1	-1	5
6	-1	-1	11
7	-1	-1	18

Null pointers

Hash Table

- Hashing allows stored data to be accessed very quickly without the need to search through every record. This is achieved by relating the data itself to its index position using a key. There are several hashing algorithms that can achieve this.
- If the calculated number is bigger than the length of the list then you will need to apply the modulo.
- Collisions occur when a bin is already occupied. In such a situation the data are placed in the next available bin.
- You can rehash with a higher modulus and number of elements when the number of collisions becomes high.
- The load factor is the number of occupied bins divided by the number of total bins.
- The hash table should contain more bins than there are elements that you would like to store by a load factor of 0.75
- If the load factor is exceeded, we can rehash using a larger hash table with a greater number of bins.

Worked example
Put the numbers 81, 93, 76,51,17, 61 into a hash table with 10 elements. Because the values are bigger than the length of the list, we apply the modulo which is the length of the table.

81 MOD 10 = 1 (81 goes into index position 1)
93 MOD 10 = 3
76 MOD 10 = 6
51 MOD 10 = 1 (a collision has occurred, place in next available position)
17 MOD 10 = 7
61 MOD 10 = 1

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	81	51	93	61		76	17		

Other hashing algorithms
If the data you want to convert has letters and not numbers you can convert the data to corresponding ASCII values.

Worked example
Let us consider the following names: Bart, Homer, Lisa, Milhouse, Ralf. We have a hash table with 10 elements.

Homer	72 + 111 + 109 + 101 + 114	507 MOD 10	7
Bart	66 + 97 + 114 + 99	393 MOD 10	3
Lisa	76 + 105 + 115 + 97	393 MOD 10	3 (collision)
Milhouse		898 MOD 10	8
Ralf		389 MOD 10	9

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
			Bart	Lisa			Homer	Milhouse	Ralf

Dictionaries

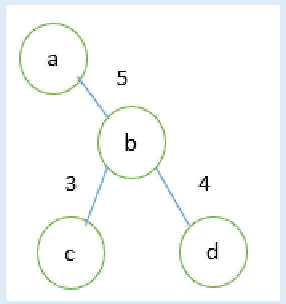
A dictionary is a abstract data type. It contains a list of pairs of values with a key that is associated with a value. We use key to access a value.

```
dict = {key1: value1, key2: value2, …, keyN: valueN}
```

Create empty dictionary	`id={}`
Create a dictionary	`id={23:"James",25:"Thomas",18:"Gordon",32:"Percy"}`
Return a value associated with a key	id[23] -> James
Add a value	id[33] = "Trevor"
List values	id
Remove a value	del id[32]

Using a dictionary to represent a graph

g = {"a":{"b":5}, "b":{"a":5,"c":3,"d":4}, "c":{"b":3}, "d":{"b":4}}



Vectors

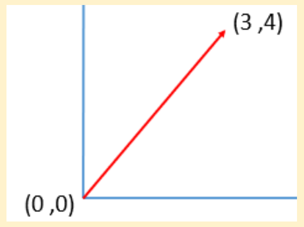
Vector notation

Function Representation
A vector can be represented as a Function (f: S → R) where S is the set that maps to R. For instance S=[0,1,2,3,4] and R=[4.0,5.5,6.7,9.1,-2.3]
0 → 4.0
1 → 5.5
2 → 6.7
3 → 9.1
4 → -2.3

List/1-D array representation
e.g. A 5 vector over R would be: [4.0,5.5,6.7,9.1,-2.3]

Dictionary representation
A 5 vector could be represented as a dictionary with both set and mapping
e.g. R={0: 4.0,1: 5.5,2: 6.7,3: 9.1,4: -2.3}

Visualisation of a vector
We can represent a vector as geometric point in space. A 2-vector e.g. [3,4] can be represented by an arrow with its tail at [0,0] and its head at [3,4]. Vectors have both magnitude and direction.



$$magnitude = \sqrt{3^2 + 4^2}$$

$$direction = \tan^{-1}(4/3)$$

Vector addition
Each element in the vector is added to the corresponding value at that element in the other vector.

Worked example
Find `a+b` where `a=[2,3,6,8]` and `b=[3,1,4,5]`

```
a = [2,3,6,8]
     + + + +
c = [3,1,4,5]
a+b = [2+3,3+1,6+4,8+5]
a+b = [5,4,10,13]
```

Scalar vector multiplication
Vectors can be multiplied by scalars (single numbers). Each element is multiplied by the scalar

Worked Example
Find `2a` where `a= [2, 3, 6, 8]`

```
2a = [(2x2), (3x2), (6x2), (8x2)]
2a = [4,    6,    12,    16]
```

Dot product
The dot product of two vectors is calculated by multiplying the corresponding element in both vector and adding together all the elements. Given vector a and b such that $a = [a_1, a_2, …, a_n]$ and $b = [b_1, b_2, …, a_n]$
Then $a.b = (a_1 \text{ x } b_1) + (a_2 \text{ x } b_2) + …. + (a_n \text{ x } b_n)$

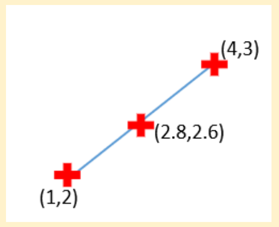
Worked example
Find `a.b` where `a= [2, 3, 6, 8]` and `b= [3,1,4,5]`

```
a      =[    2,     3,     6,     8]
             x      x      x      x
b      =[    3,     1,     4,     5 ]
a.b    =[    6 +    3 +    24 +   40]
a.b    = 73
```

Convex combination of 2 vectors
Every convex combination of 2 points lines on a line between the two points 2 points. This has the form $au + bv$ where $a + b = 1$ and $a, b >= 0$

Worked Example
Find the convex combination $au + bv$ of vectors `u=[1,2]`and `v=[4,3]`, where `a=0.4` and `b=0.6`

```
au = [1*0.4,2*0.4]
au = [0.4,0.8]
bv = [4*0.6,3*0.6]
bv = [2.4, 1.8]
au+bv = [2.4+0.4,0.8+1.8]
au+bv = [2.8, 2.6]
```



Angle between 2 vectors
The angle between 2 vectors is calculated as:

$$\cos(\alpha) = \text{a.b} / |\text{a}|.|\text{b}|$$

Worked example
Calculate the angle between two vectors `a=[3,4], b=[4,3]`

a.b = (3 x 4) + (4 x 3) = 24
$|a| = \sqrt{3^2 + 4^2} = 5$
$|b| = \sqrt{4^2 + 3^2} = 5$
24 / 5 . 5 = 24/25 = 0.96 = 16.3°