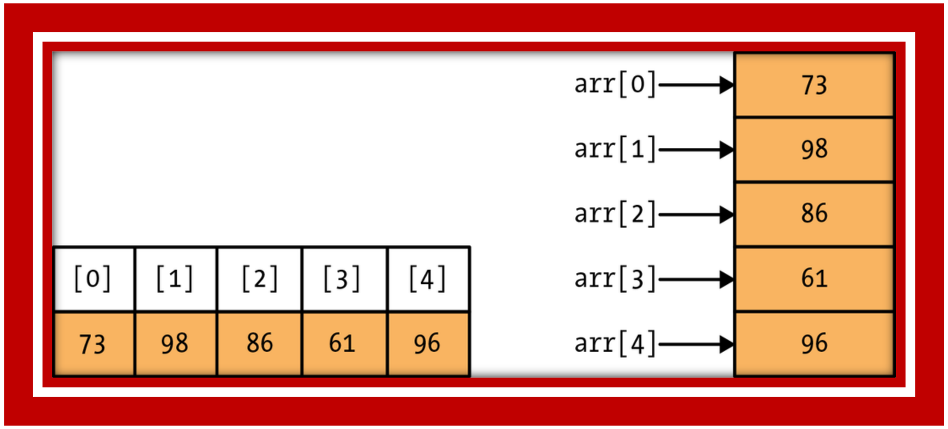
Component 1 | 1.4.2| Data structures

Specification & learning objectives

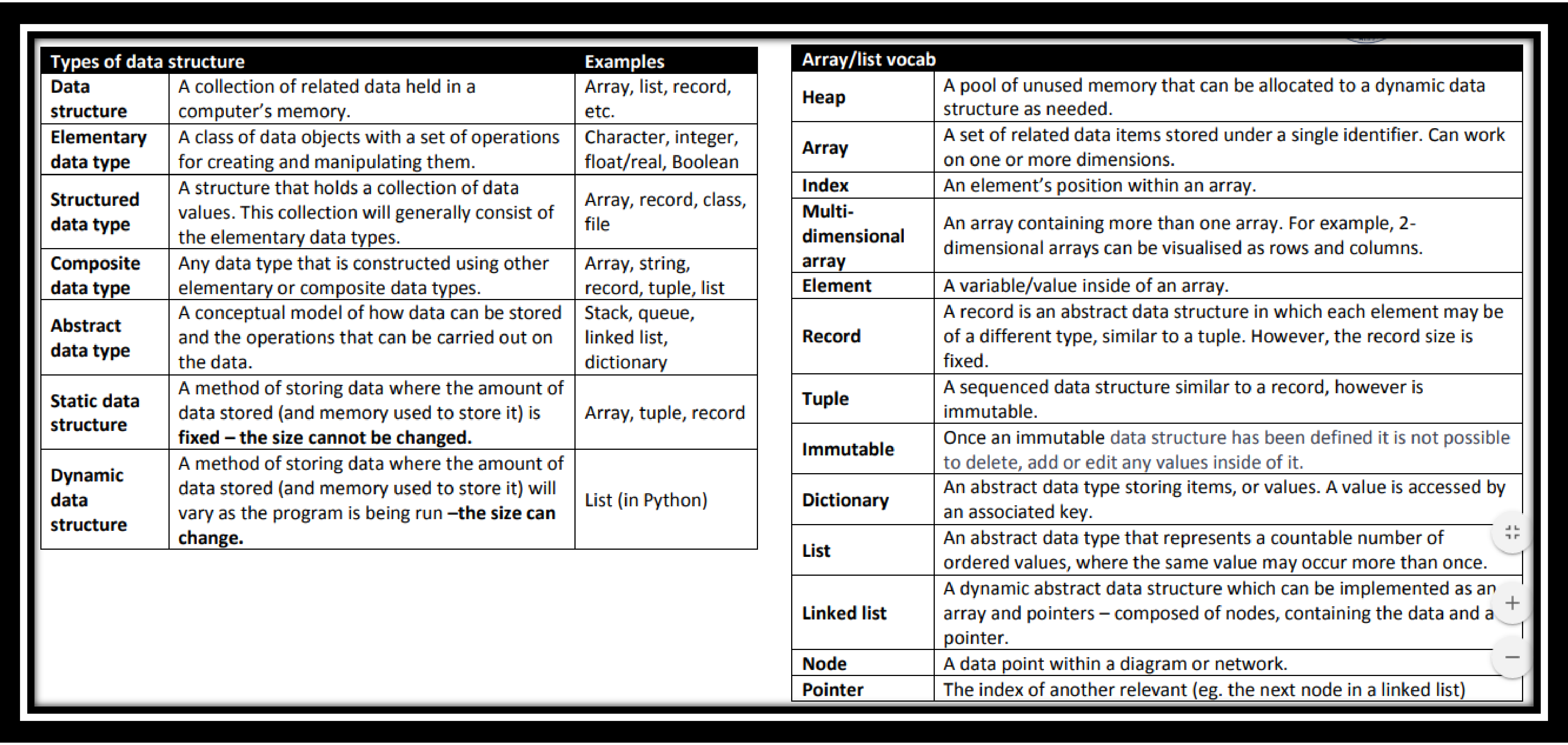
	A Level	Specification point description
	1.4.2a	Arrays (of up to 3 dimensions), records, lists, tuples
		The properties of stacks and queues
	1.4.2b	The following structures to store data: linked list, graph (directed and undirected), stack, queue, tree, binary search tree, hash table
	1.4.2c	How to create, traverse, add data to and remove data from the data structures mentioned above (This can be **either** using arrays and procedural programming **or** an object-oriented approach)

Resources

PG Online textbook page ref: 179-221
Hodder textbook page ref: 156-173
CraignDave videos for SLR 14



Component 1 | 1.4.2| Data structures

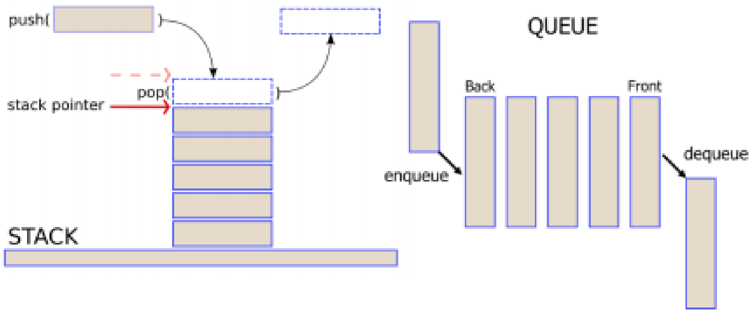


Types of data structure		Examples
Data structure	A collection of related data held in a computer's memory.	Array, list, record, etc.
Elementary data type	A class of data objects with a set of operations for creating and manipulating them.	Character, integer, float/real, Boolean
Structured data type	A structure that holds a collection of data values. This collection will generally consist of the elementary data types.	Array, record, class, file
Composite data type	Any data type that is constructed using other elementary or composite data types.	Array, string, record, tuple, list
Abstract data type	A conceptual model of how data can be stored and the operations that can be carried out on the data.	Stack, queue, linked list, dictionary
Static data structure	A method of storing data where the amount of data stored (and memory used to store it) is **fixed – the size cannot be changed.**	Array, tuple, record
Dynamic data structure	A method of storing data where the amount of data stored (and memory used to store it) will vary as the program is being run –**the size can change.**	List (in Python)

Array/list vocab	
Heap	A pool of unused memory that can be allocated to a dynamic data structure as needed.
Array	A set of related data items stored under a single identifier. Can work on one or more dimensions.
Index	An element's position within an array.
Multi-dimensional array	An array containing more than one array. For example, 2-dimensional arrays can be visualised as rows and columns.
Element	A variable/value inside of an array.
Record	A record is an abstract data structure in which each element may be of a different type, similar to a tuple. However, the record size is fixed.
Tuple	A sequenced data structure similar to a record, however is immutable.
Immutable	Once an immutable data structure has been defined it is not possible to delete, add or edit any values inside of it.
Dictionary	An abstract data type storing items, or values. A value is accessed by an associated key.
List	An abstract data type that represents a countable number of ordered values, where the same value may occur more than once.
Linked list	A dynamic abstract data structure which can be implemented as an array and pointers – composed of nodes, containing the data and a pointer.
Node	A data point within a diagram or network.
Pointer	The index of another relevant (eg. the next node in a linked list)

Key question: What are the uses of stacks and queues, and how do they work?

Stacks and queues vocab	
Queue	An abstract data structure where the first item added is the first item removed (FIFO).
Circular queue	A linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
Priority queue	Like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority.
First In, First Out (FIFO)	Where the first item added is the first item removed (FIFO).
Enqueue	A procedure for adding an item of data to the back of a queue.
Dequeue	A procedure for removing an item of data from the front of a queue.
isEmpty	A function used to check whether a queue has a size of 0.
isFull	A function used to check whether a queue's size is equal to the maximum size it was initialised with.
Append	Adding an element into an array.
Push	Adding an element onto the top of a stack.
Pop	Removing an element from the top of a stack.
Stack	An abstract data structure where the last item added is the first item removed (LIFO).
Last In, First Out (LIFO)	Where the last item added is the first item removed (LIFO).
Overflow	Occurs when trying to push more items onto a stack than it can hold.
Underflow	Occurs when trying to pop an item from an empty stack.
Call stack	A stack data structure that stores information about the active subroutines of a computer program.
Stack frame	The collection of all data on the stack associated with one subprogram call. Includes the return address, argument variables passed on the stack, and local variables.
Parameter	A specific kind of variable used to pass information between functions or procedures.
Return address	The location directly after where a subroutine is called. When a return statement is called in a subroutine or the subroutine completes the program goes to the return address and continues running the program.



```
function isEmpty
     if size == 0 then
          return True
     else
          return False
     endif
endfunction
```

```
procedure enqueue(newItem)
     if size == maxSize   then
          print ("Queue full")
     else
          rear = (rear + 1) MOD max
          q[rear] = newItem
          size = size + 1
     endif
endprocedure
```

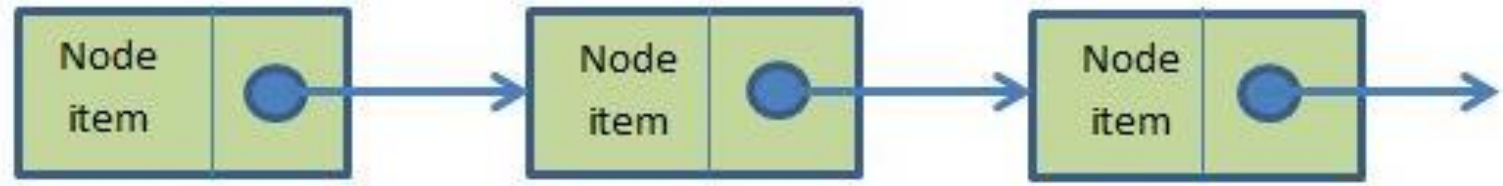
```
function isFull
     if size == maxSize then
          return True
     else
          return False
     endif
endfunction
```

```
procedure dequeue(item)
    if q.isEmpty() then
              print (“queue empty”)
    else
              q.pop(0)
    endif
endprocedure
```

Key question: How do linked lists work?

Linked lists are a valuable programming tool. They allow you to add even more structure than just ordering a list.

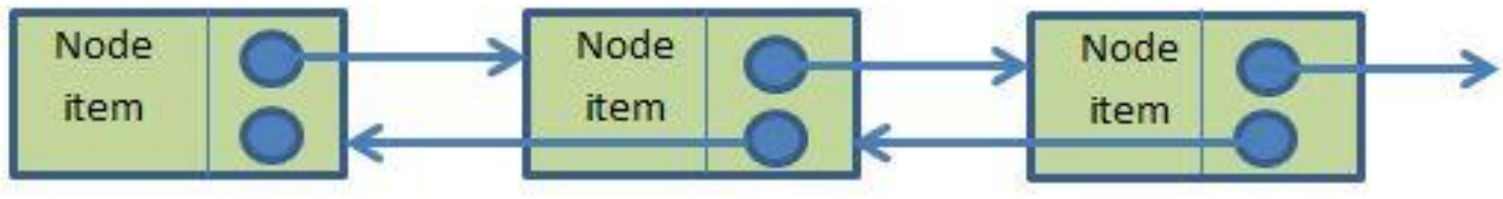
Every element (or *node*) in a linked list points to a neighbouring node. This means if the program accesses any node, it can then find its linked node. For example:



This is a *singly-linked list.* Each node in the above diagram has two parts. The node item itself, and a "pointer". The pointer tells the program where the next associated node is located.

Key question: How do linked lists work?

A doubly-linked list has two pointers. One references the location of the next node and the other points to the previous node. If the 'previous node' is null, then that is the start of the list. If the 'next pointer' is null, then that is the end of the list.



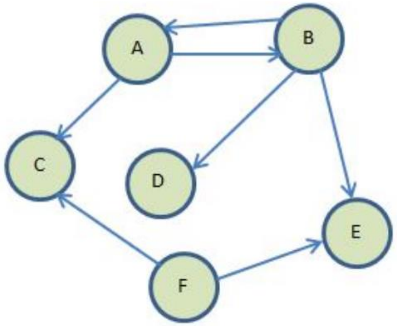
These pointers and the links they create make it simple to insert or delete items from a list without breaking the order. If an item is added or removed, the pointers of its neighbours are updated. They also make it simple for a program to traverse the list.

Linked lists are the basis of much more complicated structures such as stacks, queues and trees…

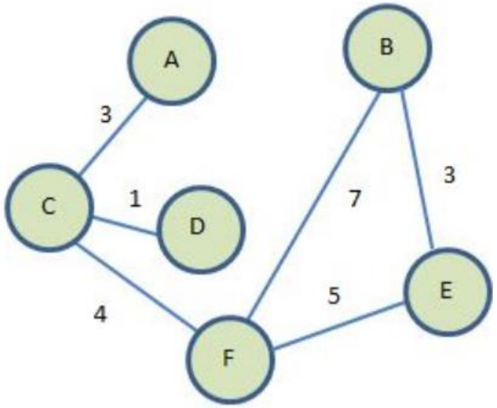
Key question: How do graphs work with breadth and depth searches?

A **graph** is a dynamic data structure for *modelling connections or relationships between items*. A graph is dynamic because it can grow and shrink at runtime.

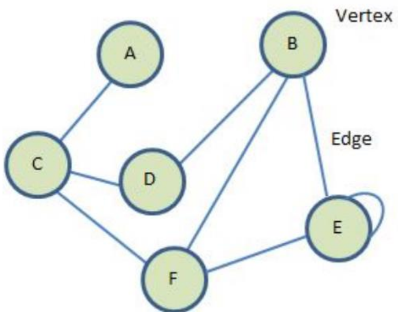
Directed graph



Labelled or weighted graph



Undirected graph

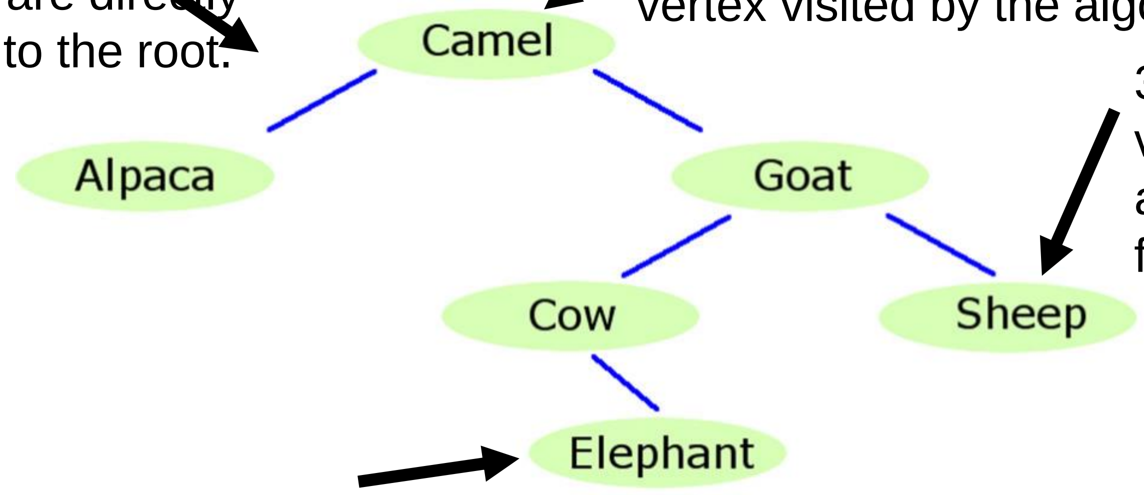


Key question: How do graphs work with breadth and depth searches?

The movement of a program through a graph is called **traversal**. There are several types of traversal, each following a different algorithm.

Breadth first traversal

- Designate one vertex as the 'root', where the traversal begins.
- Visit each vertex connected to the root.
- Once all directly connected vertices are visited, visit vertexes one step farther away.



1. **Camel** is designated as the root. It is the first vertex visited by the algorithm.

2. It then visits vertices **Alpaca** and **Goat**, as they are directly connected to the root.

3. It then visits vertices **Cow** and **Sheep**, as they are one step farther from the root.

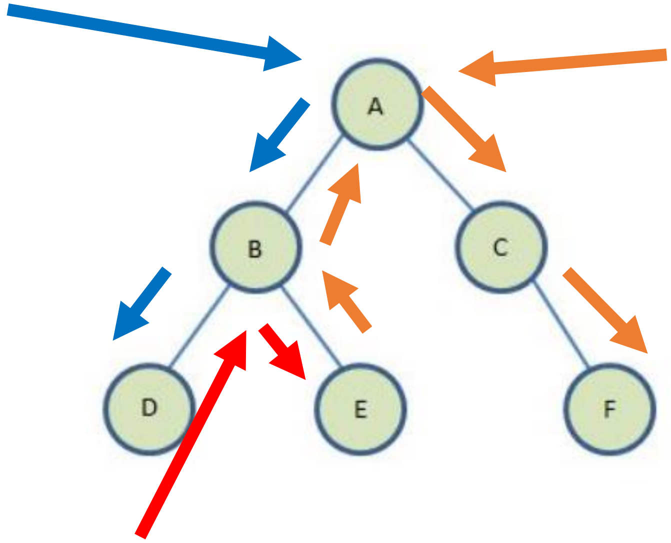
4. Finally it visits the vertex **Elephant**, as it is the most separated from the root.

Key question: How do graphs work with breadth and depth searches?

Depth first traversal

- This algorithm picks one path and follows it to the end.
- It then backtracks and follows the next path to its end.
- It will continue this until all vertices have been visited.

1. A → B → D



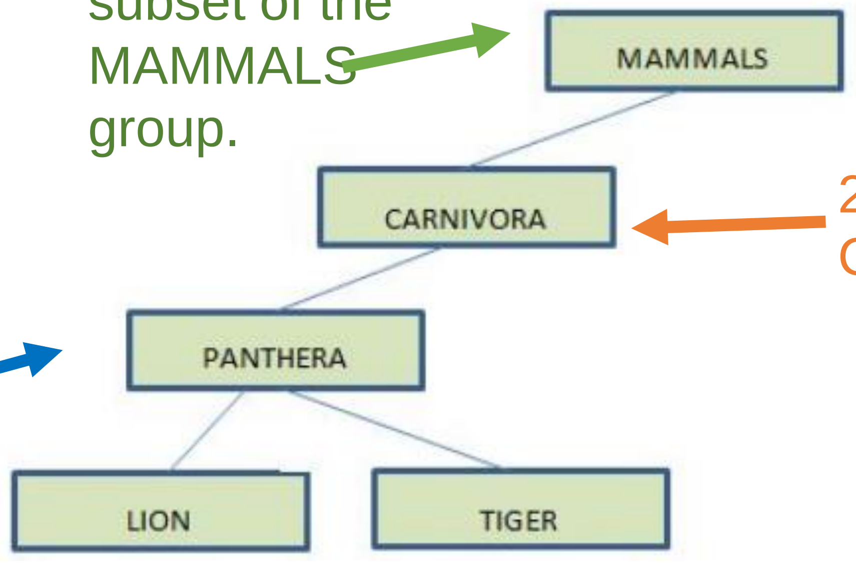
3. Backtrack **2 steps** → C → F

2. Backtrack **1 step**

Key question: How do trees and binary trees work?

In the data structures we have discussed so far, no one piece of data is more important than another. Stacks and queues limit access to all but one or two items, but overall each item simply points to the next in line.

The **tree** data structure is different. Trees use **hierarchy** to organise data, with data in lower 'branches' connected to data above and below.



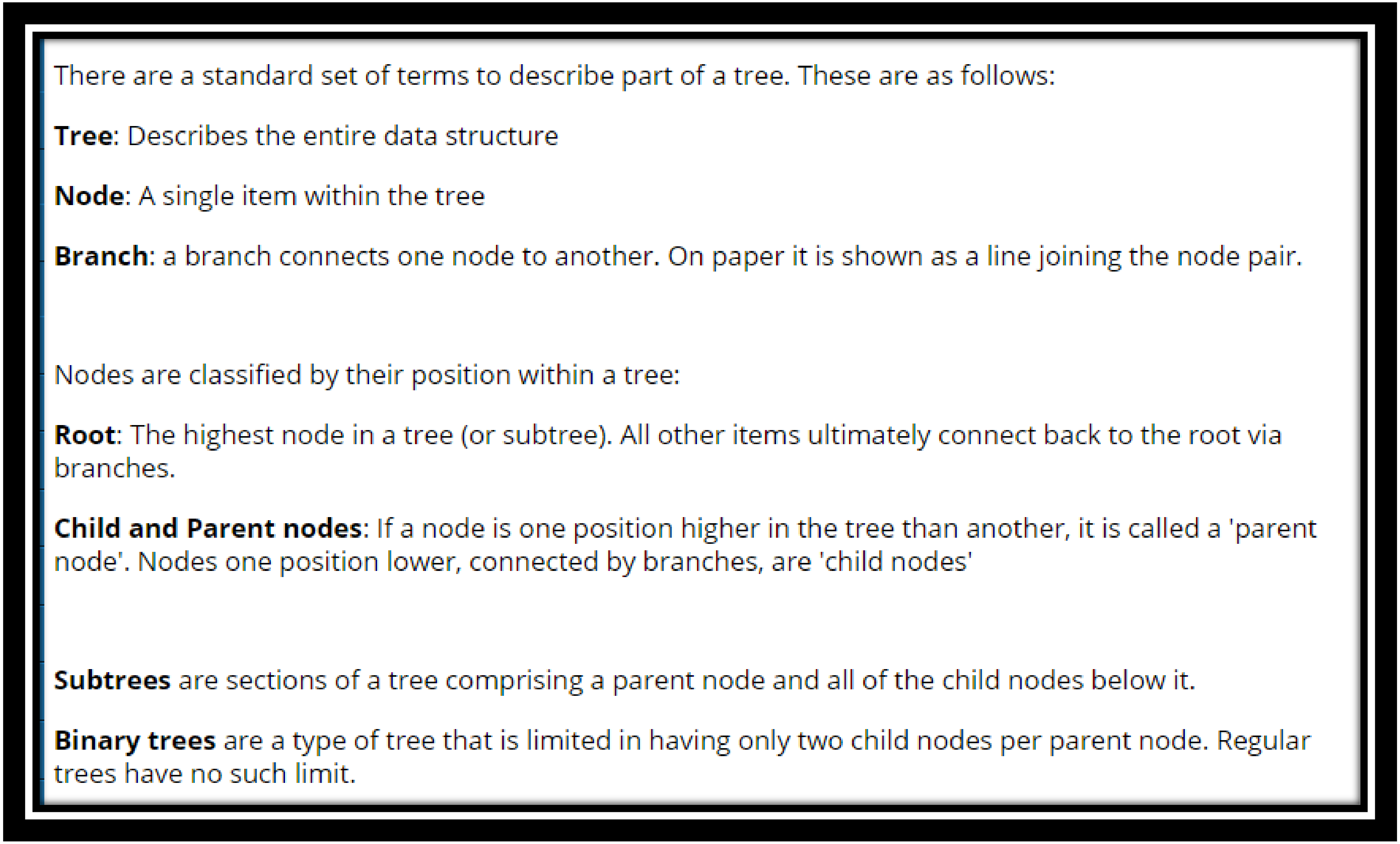
3. which is a subset of the MAMMALS group.

2. which is a subset of CARNIVORA

1. LION and TIGER are subsets of PANTHERA

Trees are useful tools in that they allow for easy **searching** on the data they contain. Database systems make extensive use of trees to index their data.

Key question: How do trees and binary trees work?



There are a standard set of terms to describe part of a tree. These are as follows:

Tree: Describes the entire data structure

Node: A single item within the tree

Branch: a branch connects one node to another. On paper it is shown as a line joining the node pair.

Nodes are classified by their position within a tree:

Root: The highest node in a tree (or subtree). All other items ultimately connect back to the root via branches.

Child and Parent nodes: If a node is one position higher in the tree than another, it is called a 'parent node'. Nodes one position lower, connected by branches, are 'child nodes'

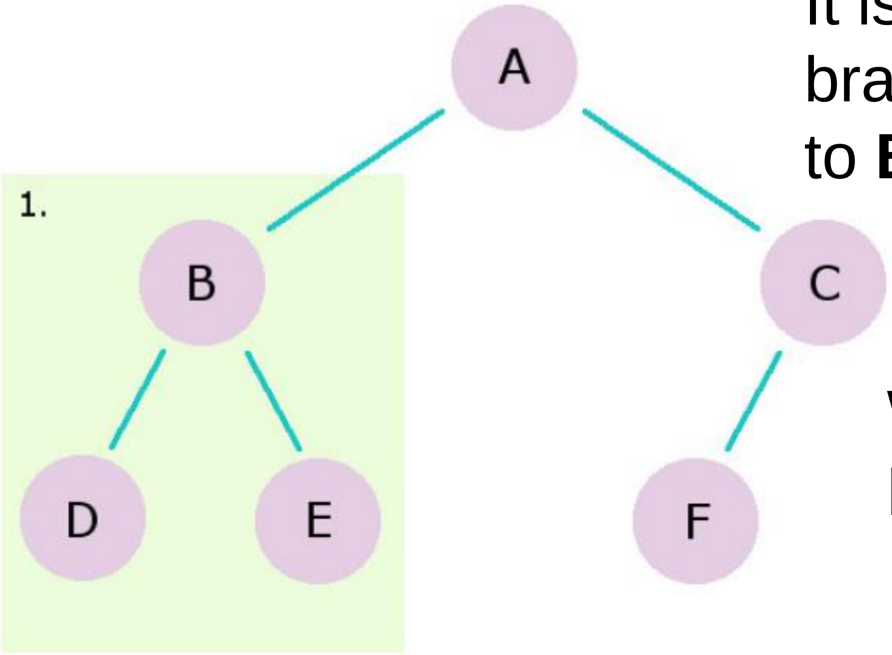
Subtrees are sections of a tree comprising a parent node and all of the child nodes below it.

Binary trees are a type of tree that is limited in having only two child nodes per parent node. Regular trees have no such limit.

Key question: How do trees and binary trees work?

A is the root node. It is connected via branches to **B** and **C**.

B is a parent node to its child nodes **D** and **E** .



while **C** is the parent node to **F**.

The area **1.**, highlighted in green, could be labelled as a sub-tree within the tree, Key question: How do hash tables work?

Hash table is a type of data structure. It allows data of any size to be mapped with a related but fixed size form.

For example a certain hashing function might map its input data into a fixed 32 bit number. This allows searching and comparing to be faster and more efficient.

For example, imagine a group of data items, where each item is a copy of the text of the novel 'War and Peace', but with a single random word in the text changed.

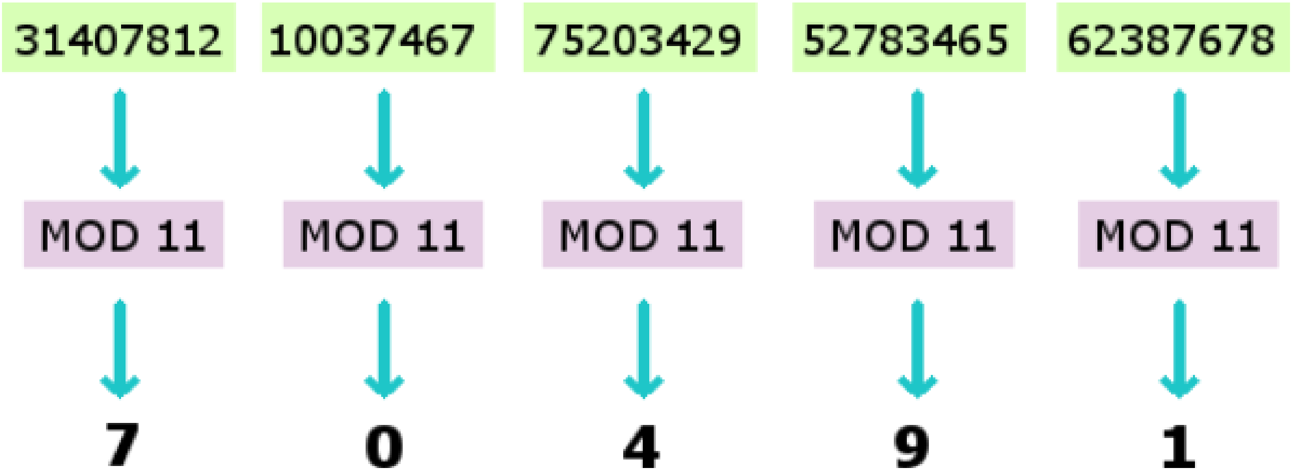
You can store the copies in a list or an array. But if you later want to run a search looking for a particular copy, the search will be very, very inefficient and slow.

This is where Hashing comes in – they can reduce the search time of even massive data sets to something manageable.

Key question: How do hash tables work?

These hashes are created by applying a **hash function** to the data items.

This example will show a simple hash function using a Modulo 11 calculation. This divides each number by 11 as many times as it will fit, then records the remainder.



31407812 10037467 75203429 52783465 62387678

MOD 11 MOD 11 MOD 11 MOD 11 MOD 11

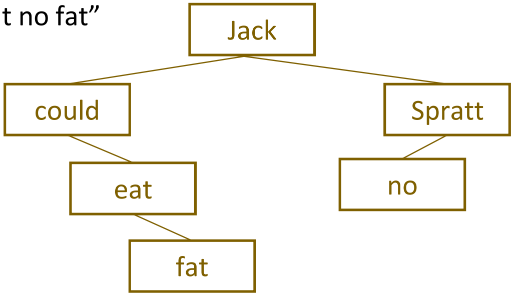
7 0 4 9 1

Running the hash function on the data set produces the following set of hashes:

Typical exam questions

1. Describe how adding new items differs between a stack and a queue. **[2]**

A tree structure storing the sentence “Jack Spratt could eat no fat” with the algorithm for insertion into a tree:



1. If tree is empty enter data item at root and stop.
2. Current node = root.
3. Repeat steps 4 and 5 until current node is null.
4. If new data is less than value at current node go left, else go right.
5. Current node = node.
6. Create new node and enter data.

2. Using this algorithm show the steps which would be taken to add the word “and” to the tree: **[5]**

3. This tree structure could be stored in either an array or a linked list. Describe **one** main different between an array and a linked list. **[2]**

Component 1 | 1.4.2| Data structures

Target: □ Overall grade: □

Minimum expectations & learning outcomes

□	Terms 175-186 from your A Level Key Terminology should be included and formatted.
□	You must explain records, lists, tuples, arrays of 1-3 dimensions and the difference between static and dynamic data structures.
□	You must include a series of annotated diagrams which clearly show a representation of stack and queue, including operations of pushing and popping and pointers. Consider overflow and underflow.
□	You must include a series of annotated diagrams which show a representation of the data structures: linked list, binary tree, graph and hash table.
□	You must illustrate how data is found, added and deleted from a linked list, binary tree, graph and hash table using both arrays and object approaches.
□	Answer the exam questions.

Feedback

Breadth	Depth	Presentation	Understanding
□ All	□ Analysed	□ Excellent	□ Excellent
□ Most	□ Explained	□ Good	□ Good
□ Some	□ Described	□ Fair	□ Fair
□ Few	□ Identified	□ Poor	□ Poor

Comment & action required

Component 1 | 1.4.2| Data structures

Reflection & Revision checklist

Confidence	Clarification
☹ 😐 ☺	Candidates should be able to describe what is meant by arrays (up to 3 dimensions), records, lists and tuples.
☹ 😐 ☺	Candidates are expected to be able recognise when they can be used and incorporate them in their programs to store data.
☹ 😐 ☺	Candidates should have an understanding of the purpose and use of a record structure to store data of different data types in a program.
☹ 😐 ☺	Candidates should have experience of using records to store, search, manipulate and retrieve data.
☹ 😐 ☺	Candidates should have an understanding of the purpose and use of a list to store data in a program.
☹ 😐 ☺	Candidates should have experience of using lists to store, search, manipulate and retrieve data.
☹ 😐 ☺	Candidates should have an understanding of the purpose and use of tuples to store data in a program.
☹ 😐 ☺	Candidates should have experience of using tuples to store, search, manipulate and retrieve data.
☹ 😐 ☺	Candidates need to have an understanding of the behaviour of stacks and queues (i.e. LIFO and FIFO).
☹ 😐 ☺	Candidates need to have an understanding of the behaviour of linked-lists, graphs, stacks, queues, trees, binary search trees and hash tables.
☹ 😐 ☺	Candidates need to be able be aware of how the aforementioned data structures can be implemented. We would recommend a general understanding of these principles that can be applied to a given scenario rather than trying to memorise code patterns.
☹ 😐 ☺	Candidates should have experience of implementing these structures in a variety of contexts, for example through a procedural program, through a different data structure and through an object-oriented approach.
☹ 😐 ☺	Candidates need to be able to read, trace and write code to implement features of these data structures. (Again we would recommend a general understanding backed up with practice implementing them, rather than trying to memorise code patterns).