Basic Data Structures

- A **data structure** is a collection of data that is organised to allow efficient processing.

Arrays / Lists

One-dimensional Arrays

- An array can be thought of like a variable that can contain more than one data item - for example storing a list of names.
- We can do this by allocating a contiguous part of memory for the purpose of storing that data.
- Contiguous means all the data is stored together, one element after the other.
- Arrays and lists are mutable, meaning that the values of the array can be changed while the program is running, just like the contents of a variable.
- Lists are different to arrays because lists are not contiguous.
- Arrays use an index relative to its start point to allow us to easily access the array's content.
- Arrays are a static data structure, meaning you can't change the size of an array once you have it set up.

Two-dimensional Arrays

- Two-dimensional arrays can be visualised as a table with two set of indexes, one for the rows and another for the columns.
- Arrays and lists only support a single data type (homogeneous).

Record Data Structure

- A record structure is a collection of related fields. A field is a variable, and each field in a record can have a different data type.
- Different data types can be used in a record.
- Records can be created with three steps:
    1. Define the record structure - what fields will be in the record?
    2. Declare a variable or array to use with the record structure.
    3. Assign and retrieve data from the variable record.

Lists and Tuples

- Both lists and tuples share many similar concepts with arrays, with a few key differences.
- Tuples are lists that can't be changed once created - they are a fixed size, set at the point of creation and said to be an immutable data structure.
- Lists, on the other hand, can be changed after they have been created - for example, you can add or remove items. They are not a fixed size but, rather, can grow or shrink in size. Lists are said to be a mutable data structure.

Stacks

- Stacks are a last in, first out data structure.
- This means that the last item of data entered is the first to be removed.
- A stack works like a stack of papers waiting to be filed.
- Items are pushed onto the top of the stack when they are added to it and popped off the top when they are deleted from it.
- You can also peek at the top item without deleting it.
- Items can only be added or removed from the top of the stack.
- To rephrase the earlier statement, the last item to be pushed onto the stack must also be the first item to be popped off.
- A stack also has a stack pointer that always points to the node at the top.
- Any attempt to push an item onto a full stack is called a stack overflow.
- Conversely, any attempt to pop an item off of an empty stack is called a stack underflow.
- Both of these potential outcomes should be considered before attempting to push or pop an item.

Operations on Stacks

- **Push: Adding an item to the top of the stack**
- **Pop: Removing an item from the top of the stack**
- **Peek: Returning the value from the top of the stack without removing it**

Implementation of Stacks

- A stack is often implemented using an array but can also be created using object-orientated techniques.

Adding an Item to a Stack (push)

- Check for stack overflow. Output an error if the stack is full.
- Increment the stack pointer.
- Insert the new data item at the location pointed to by the stack pointer.

Removing an Item from a Stack (pop)

- Check for stack underflow. Output an error if the stack is empty.
- Copy / output the data item pointed to by the pointer.
- Decrement the pointer.

Removing an Item from a Stack (pop) (OOP)

- Check for stack underflow. Output an error if the stack pointer does not pointer to a node.
- Output the node pointed to by the stack pointer.
- Set the stack pointer to the previous item.

Applications of Stacks

- Stacks are used by processors to track the flow of programs.
- When a procedure or function (subroutine) is called, it changes the value of the program counter to the first instruction of the subroutine.
- When the subroutine ends, the processor must return the program counter to its previous value - we can achieve this using a stack, which allows subroutine calls to be nested.
- Stacks are also used for:
    - Performing depth-first searches on graph data structures.
    - Keeping track of user inputs for undo operations.
    - Backtracking algorithms
    - Evaluating mathematical expressions without brackets, using a shunting yard algorithm and reverse polish notation.

Queues

- A queue is a linear data structure.
- Items are enqueued at the back of the queue (top) and dequeued from the front (bottom) of the queue.
- It is also possible to peek at the front (bottom) item without deleting it.
- A queue is like a shopping queue, the customer at the front of the queue is served first, and new customers join at the back.
- By nature, a queue system also allows for people to jump the queue - in computer science this is called a priority queue.
- In special circumstances, new items can join at the front of the queue.
- A queue is known as a first in first out (FIFO) data structure.
- A queue has a back pointer (top) that always points to the last item, sometimes called a tail pointer.
- A queue also has a front (bottom) pointer that always points to the first item, sometimes called a head pointer.
- An attempt to enqueue an item in a full queue is called a queue overflow.
- Meanwhile, trying to dequeue an item from an empty queue is called a queue underflow.
- Both of these outcomes should be considered before attempting to enqueue or dequeue an item.

Operations on Queues

- **Enqueue: Adding an item to the back of the queue**
- **Dequeue: Removing an item from the back of the queue**
- **Peek: Returning the value from the front of the queue without removing it**

Applications of Queues

- Process scheduling
- Transferring data between processors and printer spooling.
- Performing breadth-first searches on graph data structures.

Implementing Queues

- Queues can be implemented using an array or object-orientated technique.
- However, implementing a queue using an array creates a problem (see circular queues). Since both the back and front pointers are moving in the same direction as items are added and removed from the queue, the array will quickly run out of space.
- The solution is a circular queue.

Adding an Item to a Queue (enqueue)

- Check for queue overflow. Output an error if the queue is full.
- Increment the back / tail pointer.
- Insert the new data item at the location pointed to by the back/tail (top) pointer.

Adding an Item to a Queue (enqueue)(OOP)

- Check for queue overflow. Output an error if no free memory is available.
- Create a new node and insert data into it.
- The back pointer is set to point to the new node.
- If this is the first node in the list, the front pointer is set to point to the new node.

Removing an Item From a Queue (dequeue)

- Check for queue underflow. Output an error if the queue is empty.
- Copy / output the data item pointed to by the front / head pointer.
- Increment the front / head (bottom) pointer.

Removing an Item From a Queue (dequeue)(OOP)

- Check for queue underflow. Output an error if the front pointer does not point to a node.
- Output the node pointer to by the front pointer.
- Set the front pointer to the previous item.

Circular Queues

- Circular queues are where the back pointer is cycled to the front of the array when it reaches the end.
- This implementation is called a circular queue, and is not necessary when implementing a queue using object-orientated techniques.
- An array with a circular queue is ideal if you want to restrict the number of items and have a known memory footprint.

Linked Lists

- Linked lists are a data structure that provides a foundation on which other structures can be built, such as stacks, queues, graphs and trees.
- A linked list is constructed from nodes and pointers.
- A start pointer identifies the first node.
- Each node contains data and a pointer to the next node.
- Data in lists can be stored anywhere in memory, with pointers indicating the address of the next item.

Doubly Linked Lists
- By adding an extra pointer, nodes can point to the previous and next items, and these are known as doubly linked lists.

Circular Linked Lists
- A circular linked list can be created by making the last node in the list point to the first node.

Doubly Circular Linked List
- A doubly circular linked list is created by making the last node in a doubly linked list pointer to the first node.

Implementing Linked Lists

Using an Array
- A linked list can be implemented using a static array.
- Being a static data structure, arrays are stored contiguously in memory, requiring the use of an index register to determine where a specific index is in relation to a base address.

Using OOP
- With a linked list that uses objects, any available memory address can be used to store data.
- It does not need to be adjacent, as each node points to the next in structure.
- The memory footprint of the data structure is not determined at compile time and can change dynamically at run time, (basically meaning it is a dynamic data structure.

Applications of Linked Lists
- Operating systems managing a processor to store process blocks in a ready state.
- Image viewers to switch between previous and next images.
- Music players to store tracks in a playlist.
- Web browsers, to navigate backwards and forwards.
- For hash table collision resolution as an overflow.
- Maintaining a file allocation table of linked dusters on secondary storage.

Operations on Linked Lists
- **Add - Adds a node to the linked list**
- **Delete - Removes a node from the linked list**
- **Next - Moves to the next item in the list**
- **Previous - Moves to the previous item in the list**
- **Traverse - A linear search through the linked list**

Algorithms

Adding to a Linked List
1. Check if there is free memory for a new node, and output error if not
2. Create a new node and insert data into it / insert data in the node indicated by the free storage pointer.
3. If the linked list is empty:
    a. The new node becomes the first item. Create a start pointer to it.
4. If the new node should be placed before the first node:
    a. The new node becomes the first node. Change the start pointer to it.
    b. The new node points to the second node.
5. If the new node is to be placed inside the linked list:
    a. Start at the first node.
    b. If the data in the current node is less than the value of the new node:

        i. Follow the pointer to the next node.
        ii. Repeat from step 5b until the correct position is found or the end of the linked list is reached.
    c. The new node is set to point where the previous node pointed.
    d. The previous node is set to point to the new node.
6. Update the free pointer to the next available storage space.

Removing an Item From a Linked List

1. Check if the linked list is empty and output an error if there is no start node.
2. If the first item is the item to delete, set the start pointer to point to the next node.
3. If the item to delete is inside the linked list:
    a. Start at the first node.
    b. If the current node is the item to delete:
        i. The previous node's pointer is set to point to the next node.
    c. Follow the pointer to the next node.
    d. Repeat form step 3b until the item is found or the end of the linked list is reached.
4. Update the free pointer.

Traversing a Linked List

1. Check if the linked list is empty.
2. Start at the node the start pointer is pointing to.
3. Output the item.
4. Follow the pointer to the next node.
5. Repeat from step 3 until the end of the linked list is reached.

Graphs

- A graph models relationships between objects of a certain type.
- It is a data structure that consists of a set of vertices (nodes) and edges / arcs (pointers).
- These differ from a Binary Tree because they can have more than two edges for each vertices.
- Moreover they can point to any vertex in the data structure.
- The terms vertices and edges are inherited from mathematical context.
- The pointers can either point:
  - One direction - a **Directed Graph**
  - Not specify a direction - a **Undirected Graph**
- Graphs can also be weighted, with each edge given a value representing a relationship between the vertices, like distances, for example.

OCR Definition

- Data structure consisting of collection of data nodes / vertices (1)
- Connections / edges are set between nodes / vertices (1)
- Graph (edges) can be directional or bi-directional (1)
- Graphs (edges) can be directed or undirected (1)

Weight / Cost

- Graphs can also have weights
- Such a graph is called a **Weighted Graph.**
- In a weighted graph each edge is given a value representing a relationship between the vertices, for example: the distances between two towns.
  - If we were to calculate the cost to go from a town to another this can be useful.

Uses of Graphs

- To find the shortest distance / path between two locations.
- Social networking - the vertices are individual users, the edges could represent their acquaintances.
- Transport networks - the vertices are towns, the edge weights could equal distance or journey time.
- The internet: the vertices are computers or other devices and the edges are physical / wireless connections between devices.
- The world wide web - the pages are the vertices and links are the edges.
- GPS Satnav
- Fluid Dynamics
- 3D modelling

Implementation of Graphs

Undirected Graph

Edges {(A,B), (A,C), (A,D), (B,A), (B,E), (C,A), (C,D),

(D,A), (D,C), (D,F), (E,B), (E,G), (F,D), (G,E) }

Two Methods

- There are two common methods of implementing a graph:
  - Adjacency matrix
  - Adjacency list
- Graphs are typically managed as objects and / or using Arrays.
- If we manage them as objects the typical implementation uses a dictionary and is defined as an adjacency list.
- However, we can also use arrays or the adjacency matrix.

What is a Dictionary?

- A dictionary is an abstract data structure consisting of associated pairs of items, where each pair consists of a key and a value.
- It is also called an associative array, because it deals with two sets of data that are associated with each other.
- It is analogous to a real life dictionary in that there is a word (the key) associated with a definition (the data).

Adjacency Matrix

Example of undirected graph

Aa Property	# A	# B	# C	# D	# E	# F	# G
A	0	1	1	1	0	0	0
B	1	0	0	0	1	0	0
C	1	0	0	1	0	0	0
D	1	0	1	0	0	1	0
E	0	1	0	0	0	0	1
F	0	0	0	1	0	0	0
G	0	0	0	0	1	0	0

Adjacency Matrix

Example of directed, weighted graph

Aa Property	# A	# B	# C	# F
A	0	6	0	0
B	6	0	5	8
C	0	0	0	1
F	9	8	0	0

Limits of an Adjacency Matrix

- Adjacency Matrix are usually easier to implement, however a sparse graph with many nodes but not many edges could have many empty cells and the larger the graph, the more memory space will be wasted.
- Another limit could be the size when using an array, because it is a static data structure.

Adjacency List

- This approach is more space-efficient
- A list of all the nodes is created and each node points to a list of all the adjacent nodes to which it is directly linked.

Traversal of Graphs

Breadth First Traversal

- It is a method for traversing / searching a graph, exploring the nodes closest to the starting node first before exploring the nodes closest to the starting node first before progressively exploring nodes that are further away.
- It makes use of a queue data structure, which is FIFO (First In First Out) to keep track of the vertices (nodes) visited.
    - One node at the time is selected, visited and marked.
    - The adjacent nodes visited are stored in a queue.

Steps for Breadth First Traversal

1. Set the root node as the current node.
2. Add the current node to the list of visited nodes if they are not already in the list.
3. For every edge connect to the node:
    a. If the connected node is not in the visited list, enqueue the linked node.
    b. Add the linked vertex to the visited list.
4. Dequeue the queue and set the removed item as the current node.
5. Repeat from step 2 until there are no nodes left to visit.
6. Output all the visited nodes.

Pseudocode for Breadth First Traversal

current_Vertex = root

While current_vertex != Nothing

If not visited.Contains(current_vertex) Then

Visited.Add(current_vertex)

```
End If
    For each edge in vertex
If not visited.Contains(edge.vertex) Then
Queue.enqueue(edge.vertex)
End If
Next
Current_vertex = Queue.dequeue
End While
For each vertex in visited
Output current_vertex
Next
```

Depth First Traversal

- Depth first traversal is a method for traversing / searching a graph, that starts at the chosen node and goes down one route as far as possible, before backtracking and taking the next route.
- It makes use of a stack data structure, which is LIFO (Last In First Out) to keep track of the vertices (nodes) visited:
    - Visited nodes are pushed onto the stack
    - When there are no nodes left to visit the nodes are popped off the stack.
- There is more than one valid output from a depth-first search.
- You will typically see the left-most path being traversed first in mark schemes and other sources.
- However is is perfectly valid to follow the right-most path first - or any random edge you like.
- The results will be different, but you are still performing a depth-first search - it all comes down to how you choose to implement it.

Steps for Depth First Traversal

1. Set root vertex as current vertex
2. Add the current vertex to the list of visited vertices.
3. If the vertex has an edge that has not been visited:
    a. Push the linked vertex to the stack.
    b. Repeat from step 2 until all edges have been visited.
4. Pop the stack and set the item removed as the current vertex.
5. Repeat from step 2 until there are no vertices to visit.
6. Output all the visited vertices.

Pseudocode For Depth First Traversal

```
Current_vertex = root
While current_vertex != nothing
If not visited.Contains(current_vertex) Then
Visited.Add(current_vertex)
End If
For each edge in vertex
If not visited.Contains(edge.vertex) Then
Stack.push(edge.vertex)
End If
Next
Current_vertex = Stack.pop
End While
For each vertex in visited
Output current_vertex
Next
```

Trees

- A tree is a fundamental data structure used in many areas of computer science. It consists of nodes and pointers.
- Each tree has a root node. Unlike an actual tree, when we visualise a tree data structure, the root is at the top. The root connects to zero, one or more child nodes.
- Nodes at the very bottom of the tree are called leaf nodes.
- Nodes are connected with pointers, also known as edges.
- A set of nodes and edges from any single node down through all of its descendants is known as a subtree.

Typical uses of Trees

- Storing and managing file and folder structures.
- Implementations of the A* pathfinding algorithm.
- Storing and manipulating any form of hierarchical data that requires a parent node to have zero, one or more child nodes.

Binary Trees

- A binary tree is similar to a standard tree data structure, consisting of nodes and pointers.
- It is a special case of a graph where each node can only have zero, one or two pointers, with each pointer connecting to a different node.
- Since a binary tree is essentially a graph, nodes and pointers can also be described as vertices and edges.

Implementation of Binary Trees

- Binary trees can be represented in memory with dictionaries:
- tree = {“E”:[“B”,”G”], “B”:[“A”,”C”], “G”: [“F”, “H”], “A”:[], “C”:[], “F”:[], “H”:[]}
- Binary trees can be stored as arrays.
- It is more common to see a binary tree represented as objects - nodes with a left and right pointer.
- The left and / or right pointer is set to null if there is no child node.

Adding an item to a Binary Tree

1. Check if there is free memory for a new node - if there isn’t, output an error.
2. Create a new node and insert data into it.
3. If the binary tree is empty:
    a. The new node becomes the first item. Create a start pointer to it.
4. If the binary is not empty:
    a. Start at the root node
    b. If the new node should be placed before the current node, follow the left pointer.
    c. If the new node should be placed after the current node, follow the right pointer.
    d. Repeat from step 4b until the leaf node is reached.
    e. If the new node should be placed before the current node, set the left pointer to be the new node.
    f. If the new node should be placed after the current node, set the right pointer to be the new node.

Removing an item from a Binary Tree

1. Start at the root node and set it as the current node.
2. While the current node exists and it is not the one to be deleted.
    a. Set the previous node variable to be the same as the current node.
    b. If the item to be deleted is less than the current node, follow the left pointer and set the discovered node as the current node.
    c. If the item to be deleted is greater than the current node, follow the right pointer and set the discovered node as the current node.
3. If the previous node is greater than the current node, the previous node’s left pointer is set to null.
4. If the previous node is less than the current node, the previous node’s right pointer is set to null.

If the Node has one Child

1. If the current node is less than the previous node:
    a. Set the previous node’s left pointer to the current node’s left child.
2. If the current node is greater than the previous node:
    a. Set the previous node’s right pointer to the current node’s right child.

If the node has two children

1. If a right node exists and it has a left sub-tree, find the smallest leaf node in the right subtree:
    a. Change the current node to the smallest left node.

    b. Remove the smallest leaf node.
2. If a right node exists and it has no left subtree:
    a. Set the current node to be the current node's right pointer.
    b. Set the current node's right pointer to null.

Traversing a Binary Tree

Pre-order node-left-right

1. Start at the root node
2. Output the node
3. Follow the left pointer and repeat from step 2 recursively until there is no pointer to follow.
4. Follow the right pointer and repeat from step 2 recursively until there is no pointer to follow.

in-order Traversal left-node-right

1. Start at the root node
2. Follow the left pointer and repeat from step 2 recursively until there is no pointer to follow.
3. Output the node
4. Follow the right pointer and repeat from step 2 recursively until there is no pointer to follow.

Post-Order Traversal left-right-node

1. Start at the root node
2. Follow the left pointer and repeat from step 2 recursively until there is no pointer to follow.
3. Follow the right pointer and repeat from step 2 recursively until there is no pointer to follow.
4. Output the node.

Applications of a Binary Tree

- Binary trees have many uses in computer science:
    - Database applications where efficient searching and sorting is required without moving items, which is slow.
    - Wireless networking and router tables.
    - Operating systems scheduling processes.
    - Huffman coding, used for compression algorithms like JPEG and MP3.
    - Cryptography (GMM trees).

Operations that can be Performed on a Tree

- Add: Adds a new node to the tree
- Delete: Removes a node from the tree
- Binary Search: Returns the data stored in a node
- Pre-order traversal: A type of depth-first search
- In-order traversal: A type of depth-first search
- Post-order traversal: A type of depth-first search
- Breadth-first search: Starts at the root node and visits each node at the same level before going deeper into the structure.

Hash Tables

What is a Hash Table?

- The goal with a hash table is to immediately find an item in a sorted or unsorted list without the need to compare other items in the data set.
- Programming languages use hash tables to implement a dictionary data structure.
- A hashing function is used to calculate the position of an item in a hash table.
- The hashing function is applied to an item to determine a hash value — its position in a hash table.
- A hash table needs to be at least large enough to store all the data item but is usually significantly larger to minimise the chance of the algorithm returning the same value for more than one item, known as a collision. This is bad as two data items cannot occupy the same position in the hash table.

Properties of a Good Hashing Function

- Be calculated quickly
- Result in as few collisions as possible
- Use as little memory as possible

Resolving Collisions

- There are many strategies for resolving collisions generated from hashing functions.
- A simple solution is to repeatedly check the next available space in the hash table until an empty position is found and store the item there - this is known as open addressing.
- To find the item later, the hashing function delivers the start position from which a linear search can then be applied until the item is found - this is known as linear probing.
    - A disadvantage of linear probing is that it can prevent other items from being stored at their correct location in the hash table.
    - It can also result in clustering - several positions being filled around common collision values.
- With hashing algorithms, there is often a trade-off between the efficiency of the algorithm and the size of the hash table.
- The process of finding an alternative position for items in the hash table is known as rehashing.
- An alternative method of resolving collisions is to use a two dimensional hash table.
    - It is then possible for more than one item to be placed at the same position, known as chaining.
- Another possibility would be to use a secondary table for collisions, known as an overflow table, which can then be searched sequentially.
- You could also use a linked list to maintain your overflow - again, searched sequentially.

Applications of Hash Tables

- Hash tables can be used in situations where items in a large data set need to be found quickly. Typical uses include:
    - File systems linking a file name to the path of a file.
    - Identifying keywords in a programming language during compilation.

Operations on a Hash Table

- Add: Adds a new item to the hash table
- Delete: Removes an item from a hash table
- Retrieve: Retrieves an item from a hash table using its hash value.

Hash Table Algorithms

Adding an Item to a Hash Table

1. Calculate the position of the item in the hash table using a hashing function.
2. If the calculated position is empty, insert the item and stop.
3. If the calculated position is not empty, check the first position in the overflow table.
    a. If the position is empty, insert the item and stop.
    b. Increment the position to check in the overflow table.
    c. Repeat from step 2a until the item is inserted or the overflow table is found to be full.

Removing an Item from a Hash Table

1. Calculate the position of the item in the hash table using a hashing function.
2. If the calculated position contains the item to be deleted, delete it and stop.
3. If the calculated positions does not contain the item to be deleted, check the first position in the overflow table.
    a. If the position contains the item to be deleted, delete it and stop.
    b. Increment the position to check in the overflow table.

    c. Repeat from step 3a until the item to delete is discovered or the end of the overflow table is reached.

Traversing a Hash Table

1. Calculate the position of the item in the hash table using a hashing function.
2. Check if the item in that position is the item to be found.
3. If the calculated position does not contain the item to be found, move to the first position in the overflow table.
    a. Check if the item in that position is the item to be found.
    b. If it is not, increment to the next position in the overflow table.
    c. Repeat from step 3a until the item is found or the end of the overflow table is reached.
4. If the item has been found, output the item data. If it has not, output “Not found”.