WJEC A2 Computing Unit 3.1->3.3 Cheat Sheet

by AlexHoratio (Horatio) via cheatography.com/21127/cs/16024/

Records

Made up of multiple **fields**

Each **field** consists of a single data type

Each record is processed as a single item

Binary Trees

Binary trees are tree entities wherein each element has no more than two children.

Pre-Order Traversal

1. **Root**
2. **Left**
3. **Right**

In-Order Traversal

1. **Left**
2. **Root**
3. **Right**

Post-Order Traversal

1. **Left**
2. **Right**
3. **Root**

De Morgan's Law

De Morgan's law states that **NOT(A OR B)** is the same as **NOT(A) AND NOT(B)**.

Also, **NOT(A AND B)** is the same as **NOT(A) OR NOT(B)**.

Recursion

A **recursive** function is a function that calls itself repeatedly until a certain condition is met, at which point it "collapses" and returns a result.

Data Validatio

Presence Check	This check ensures that *something* has been entered.
Range Check	Ensures that data is within a given range.
Type Check	Checks to make sure the data is of the correct type(i.e Integer, String, Float)

Data Validatio (cont)

Format Check	Ensures that the data follows a particular format (for use with Post Codes)
List Check	Ensures that data is one of a given list of options, for example "M", "F" or "X" for a gender check
Length Check	Make sure that the data is of the correct length(in terms of characters, digits or elements)

Dijkstra's Pathfinding Algorithm

1. Start from the first vertex, named **A**.

2. Work out the weight from the A to each other connected node, in this case **B** and **C**. Add these costs to a list and order the list from lowest cost to highest.

3. "Move" to the node that has the lowest cost in the list.

4. Calculate the cost to move from A to any nodes adjacent to the currently visited node, and add these costs to the list.

5. Ignore longer routes(for instance, going from A to B to C rather than A to C)

6. Repeat steps 2->5 until you arrive at your destination.

Stacks

Stacks are **LIFO**(**L**ast **I**n, **F**irst **O**ut), so items can be **pushed** or **popped** from the top of the stack

Stacks may be used for:
1. Storing return addresses
2. Storing intermediate steps in arithmetic operations
3. Interrupts

Hash Tables

A **hash table** is composed of both a **table of data** and a **key**. The key is a smaller file which identifies the location of a specific piece of data in the much larger table.

A hashing calculation is performed on each piece of data before it is placed into the table. The location it is placed at is determined via the result of the hash.

If there is a hash collision, one of two things will happen:

1. **Chaining** - This is where a list is created in the memory location where a collision has occurred and each element that is stored there becomes an element in that list.
2. **Rehashing** - This involves running another hashing algorithm on the result of the first hash to obtain a new memory location.

Data Verification

Screen Verification	This is a visual check to ensure that no errors have been made. For instance, sending the data back to the source to check over it.
Double - Keying	This is where data is entered twice to confirm i.e passwords or e-mail addresses.
Check Digit	This can be used for numerical data like credit card numbers or ISBNs, where one digit is the result of a mathematic calculation performed on the others.

Lossy Compression

Some of the data is "**lost**" during compression

Lossy data types include **JPEG**, **MPEG** or **MP3**

Queues

Queues are **FIFO**(**F**irst **I**n, **F**irst **O**ut).

These would be used for
1. Holding tasks for the computer to run
2. A keypress buffer
3. Spooling output to a disk while waiting for a printer

Graphs

Graphs can relate any number of element to any other number of elements.

A **Weighted Graph** is where there is a value on each edge which represents the "cost" of traversing it.

Graphs can either be **directed** or **undirected**.

> 1. A **directed** graph is a graph wherein there is a one-way relationship between each node.
> 2. An **undirected** graph is a graph wherein the relationship is bidirectional between each node.

Data Flow Diagrams

What it looks like	What it is
Weird Double-Square Lookin Thing	External Entity
Rounded Square	Process
Rectangle Missing the Right Edge	Data Storage
Arrow	Represents the flow of data(should be labelled)

Big-O Notation

Constant Time O(1)	No matter how much data you give this algorithm, it will always execute in the same amount of time.
Linear Time O(n)	The time taken for the algorithm to run is directly proportional to the size of the input data.
Polynomial Time O(n^2)	The time taken increases proportional to the square(or cube, or any other power greater than one) of the input size.
Exponential Time O(2^n)	The time taken will double with every additional unit(or triple, etc- the 2 can be any number greater than one) of input data.
Logarithmic Time O(Log(n))	The time taken increases logarithmically, so the rate of increasing time with respect to input size actually decreases.

Lossless Compression

No data is lost in the process, and the entire original file can be reconstructed.

Another type of lossless compression is via **Dictionary Encoding**, where the most frequent elements of data are removed and instead replaced with a "token" that points to the piece of data somewhere else in the file to prevent needless data duplication.

Also in **Huffman Coding**, the most frequent elements of data are given the shortest token to maximize compression.

One type of lossless compression is known as **Run Length Encoding**, where strings of the same data(e.g, "BBBBBBBB") are replaced with the number of times to place the data in a row when decompressing it. (8B)