1.4.2 Data Structures

Revision sheet

1 Arrays

A data structure for storing a finite, ordered set of data of the same data type within a single identifier.
Arrays come in up-to three dimensions. They are all fundamentally the same.

1.1 1-Dimensional Array

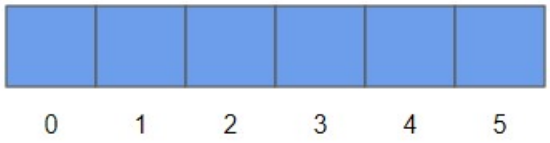

Figure 1: 1D Array

The one-dimensional array is based off of a 0 based index. The pseudocode below shows how you can set the contents of the array and how to access one of the indexes.

```
oneDimensionalArray = [1, 23, 12, 14,
↪  16, 55]
print(oneDimensionalArray[3]) //outputs
↪  14
```

1.2 2-Dimensional array

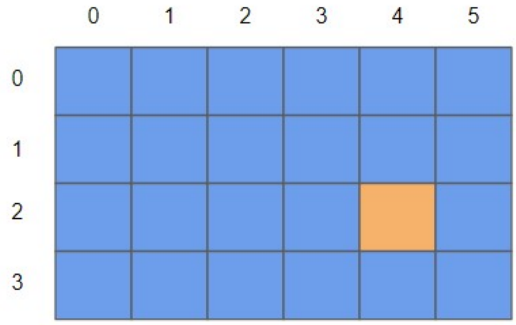

Figure 2: 2D Array

Two-Dimensional arrays are declared in a similar way to that of a 1D array.

```
twoDimensionalArray = [[1, 23, 12, 14,
↪  16, 55], [2, 56, 67, 23, 11, 89],
↪  [23, 12, 114, 64, 25, 65], [97, 65,
↪  16, 13, 36, 56]]
print(twoDimensionalArray[1,3]) //down
↪  then across. Outputs 23
```

For the diagram shown in Figure 1.2, the orange square would be at location `[2,4]` as the indexes are addressed down then across.

1.3 3-Dimensional Arrays

3D arrays can be visualised as a multi-page spreadsheet, and can be thought of as multiple 2D arrays.

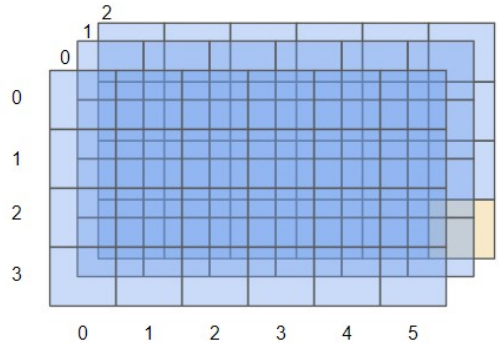

Figure 3: 3D Array

Three-Dimensional arrays are declared in much the same way as two dimensional arrays, except there are multiple 2D arrays in the same declaration.

```
threeDimensionalArray = [[[12,8],[9,6,19
↪  ]],[[241,89,4,1],[19,2]]]
print(threeDimensionalArray[0,1,2])
↪  //outputs 19
```

For the diagram shown in Figure 1.3, the orange square would be at location `[2,3,5]` as the it is [arrayNumber, rowNumber, columnNumber].

2 Tuples

A data structure for sorting an immutable (cannot be modified once created), ordered set of data, which can be of different data types, within a single identifier.
Tuples are initialised using regular brackets instead of square brackets:

```
tupleExample = ("Value1", 2, "Value3")
print(tupleExample[0]) //outputs "Value1"
```

3 Records

A data structure that stores data in elements called fields, organised based on attributes.

ID	FIRST_NAME	SURNAME
001	Dave	Andrews
002	Sally	McAnthony
003	Fred	Fredrikson

Figure 4: Record

The pseudocode below would be used to declare the record shown in Figure 3.

```
    pupilDataType = record
    integer ID
    string FIRST_NAME
    string SURNAME
end record
```

Each field in the record can be identified by `recordName.fieldName`. The variable which holds the record is declared as `pupil1 : pupilDataType`

4 Linked Lists

A data structure that stores an ordered sequence of data where each item is stored with a pointer to the next item. The items are not stored contiguously (in the same order) in memory.
Linked lists are dynamic.

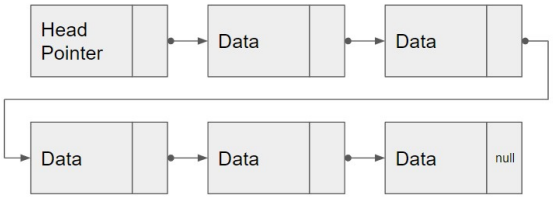

Figure 5: Linked List

Linked lists are good because they can grow according to the needs of the user and it is easier to insert a new element (you just insert it at any point in memory and update the relevant pointers). To search a linked list, you have to start at the head pointer and work through every element until you either reach the end or find the item you are looking for. This can take some time if you have a very long linked list. Due to it not being stored contiguously in memory, there is no random access, to find any elements you have to search as described above.
Linked lists could be implemented in an object oriented approach with attributes for the index (memory location), data and pointer to the next index.

5 Hash Tables

A data structure where a hashing algorithm calculates a value to determine where a data item is to be stored. The data item can then be directly accessed by recalculation, without any search.
These are often implemented as an array. A good hashing algorithm would have a low possibility of a collision occurring; if a collision does occur, it will need to be resolved. There are two ways in which collisions can be resolved.

Linear probing If the address is already full, then you move through the table 1 index at a time to find a free space. The data is put in the first free space found. Advantages: can use less memory; can perform faster; if there isn't a collision, the data is placed in the first place you look.

Chaining Each location points to a linked list. New items are inserted into the linked list. When searching, you go through the linked list to find the item. Advantage: no limit on the amount of data that can be stored.

Overflow area There is an overflow area setup at the bottom of the table, any collisions get

dumped in the overflow area. Disadvantage: really inefficient as to find an item, you might have to look through all of the overflow area, which could be massive.

6 Stack

A last-in-first-out (LIFO) data structure. The last item added/pushed is the first item to be removed/popped off.
Stacks can either be static or dynamic and are often implemented using one directional arrays, with additional pointers for current top item or next free slot, max size.

Figure 6: Stack

Variables which are in memory are loaded onto a stack when a function is called.

7 Queues

A first-in-first-out (FIFO) data structure. The first item enqueued on to the queue is the first to be dequeued off of the queue.
Queues can either be circular or linear. Queues can be implemented in an array. Alongside the array, you will need a number of pointers: front pointer, rear pointer, max number of elements, number of elements currently in the queue. Queues are static and quite often used in a keyboard buffer.

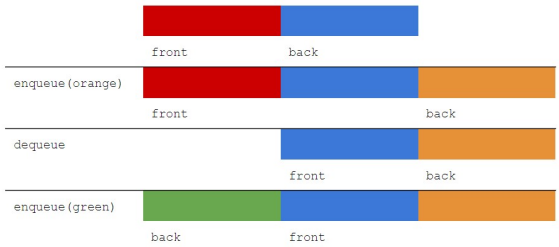

Figure 7: Circular queue

The diagram above shows how a circular queue works.

8 Graphs

A data structure consisting of a set of vertices/nodes connected by a set of arcs/edges.

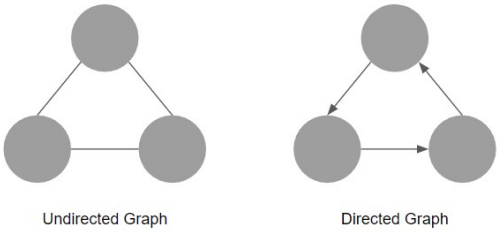

Figure 8: Undirected and directed graphs

Graphs come in two flavours, directed and undirected. The edges on a directed graph are shown with arrows, whereas on an undirected graph they are plain lines. Edges on a graph can be weighted (shown by a number on the edge).

8.1 Storing The Graph

Graphs can be stored as an adjacency matrix or adjacency list.

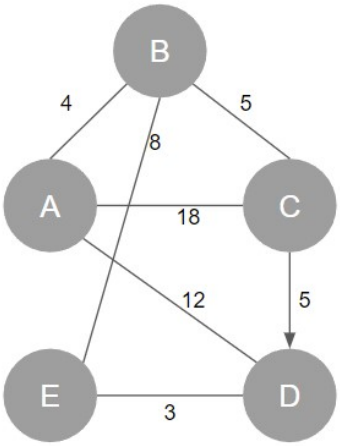

Figure 9: Graphs used for Storage Examples

8.1.1 Adjacency Matrix

Adjacency matrices can be implemented as a two dimensional array as shown below.

	A	B	C	D	E
A	-	4	18	12	-
B	4	-	5	-	8
C	18	5	-	5	-
D	12	-	-	-	3
E	-	8	-	3	-

The matrix is read down then across. For example traversing from C(down) to D(across) is 5, but traversing from C(across) to D(down) is - as C-D is directed. Adjacency matrices are more convenient to work with as they have shorter access times and are easier to add nodes.

8.1.2 Adjacency List

A B:4, C:18, D:12

B A:4, C:5, E:8

C A:18, B:5, D:5

D A:12, E:3

E B:8, D:3

Adjacency lists are more efficient for larger, sparser networks.

8.2 Traversal

Traversing a graph can either be done breadth-first or depth-first.

8.2.1 Breadth-First Traversal

Used for finding the shortest path in things like GPS systems; applications like Facebook for keeping track of friends; web crawlers.

Steps

1. Pick one of the graph vertices and put it at the back of the `toVisit` queue.
2. Take the first item off of `toVisit` and add it to the `visited` list.
3. Enqueue all children of the current node which aren't in `visited` to `toVisit`.
4. Keep repeating steps 2 and 3 until `toVisit` is empty.

8.2.2 Depth-First Traversal

This can either be pre-order or post-order.

Used for mazes and job scheduling.

Steps

1. Pick anyone of the graph vertices and push it onto the stack `toVisit`.
2. Pop the first item off of `toVisit` and add it to `visited`
3. For the current node, if its children aren't in `visited` push them to `toVisit`.
4. Keep repeating steps 2 and 3 until `toVisit` is empty.

9 Trees

A data structure that uses a set of linked nodes to form a hierarchical structure starting at a root node. Each node is a child/sub-node of a parent node.
There are two types of trees: binary and non-binary. Fundamentally, all trees are the same.

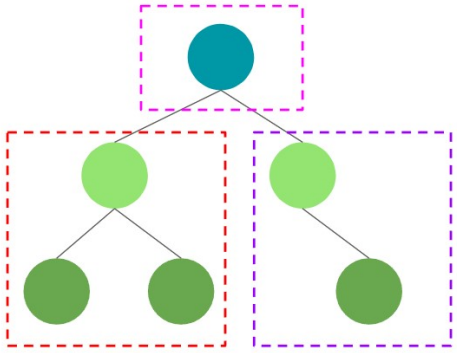

Figure 10: Basic structure of a tree

The cyan node is called the root node. The light green nodes are parent nodes. The dark green nodes are leaf/children nodes. The red box indicates the left sub tree and the purple box indicates the right subtree.
The key difference between a binary and non-binary tree are the number of child nodes a single node can have. In a binary tree, a parent node can only have 0, 1 or 2 child nodes. Whereas in a non-binary tree, a parent node can have any number of child nodes.

9.1 Traversing a Tree

There are three different *routes* which can be taken when traversing a tree.

9.1.1 In Order Traversal

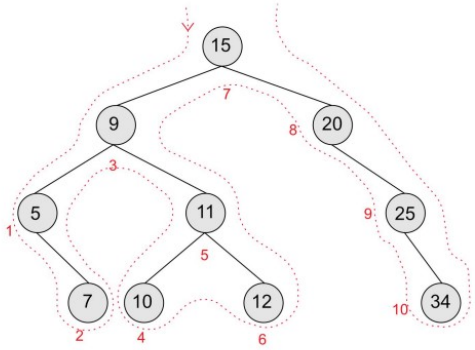

Figure 11: In order traversal route

Traverse: left subtree, root node, right subtree.

9.1.2 Pre-Order Traversal

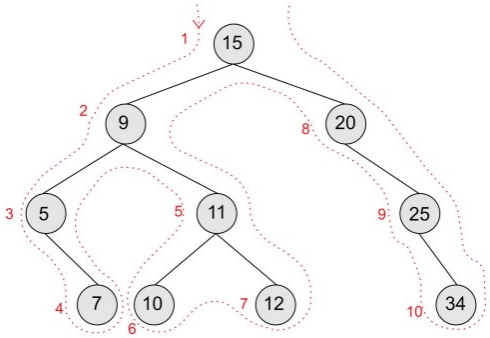

Figure 12: Pre-order traversal route

Traverse: Root node, left subtree, right subtree.

9.1.3 Post-Order Traversal

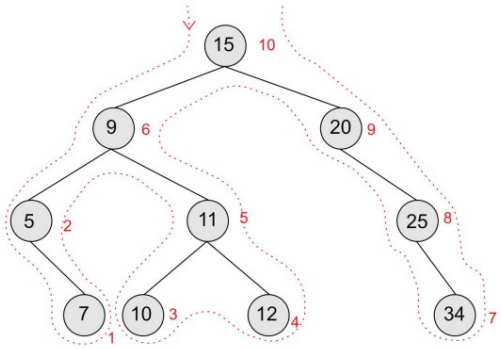

Figure 13: Post order traversal route

Traverse: Left subtree, right subtree, route node.