ZNOTES // A-LEVEL SERIES

visit www.znotes.org

Updated to 2017-19 Syllabus

CIE A2-LEVEL COMPUTER SCIENCE 9608

SUMMARIZED NOTES ON THE PRACTICAL SECTION

TABLE OF CONTENTS

4.1. Computational Thinking and Problem Solving

4.1.1 Abstraction

- Abstraction is the process of modeling a complex system in an easy to understand way by only including essential details, using:
  - Functions and procedures with suitable parameters →Imperative Programming
  - Classes → Object Orientated Programming
  - Facts and rules → declarative programming
  - ADTs (Abstract Data Types -see section 4.1.3)

4.1.2 Algorithms

- Serial/Sequential/Linear Search
  - All the values are considered in sequence
  - Even if an item is not found, all the values will have been considered
  - Best-case scenario: item to be found is at the start of the list → O(1)
  - Worst-case scenario →max number of comparisons, when item to be found is at the end of the list → O(N) where N is the number of elements in the list
  - Average number of comparisons → N/2
- Binary Search
  - Used to search an ***ordered*** array
  - Much faster than a linear search for arrays of more than a few items
    1. Ordered array divided into 3 parts: middle, lower and upper
    2. Middle item is examined to see if it is equal to the sought item
    3. If not, and the middle value is greater than the sought item, the upper part of the array is disregarded
    4. The process is repeated for the bottom part

  - Worst-case → $\log_2 N + 1$ → O($\log_2 N$)
  - When compared to linear search, whose worst-case behaviour is N iterations, we see that binary search is substantially faster as N grows large. For example, to search a list of one million items takes as many as one million iterations with linear search, but never more than twenty iterations with binary search

Recursive Binary Search

```
fruits = ["apple", "banana", "cherry", "kiwi", "lemon", "mango", "plum

Low =0
High= 7


def BinarySearch(Low, High):
    global Found
    if Low > High:
        return "Error, empty array"
    middle = int((Low+High)/2)
    if fruits[middle] == search:
        Found = middle
    elif fruits[middle] > search:
        BinarySearch(Low, middle -1)
    elif fruits[middle] < search:
        BinarySearch(middle+1, High)
    return Found

search = input("Enter a fruit you are looking for: ")
findFruit = BinarySearch(Low,High)

print(findFruit)
```

Iterative Binary Search

```
fruits = ["apple", "banana", "cherry", "kiwi", "lemon", "mango", "plu
searchFruit = input("Enter a fruit you are looking for: ")

x = 0
low = 1
high = 7

while (high>=low) and (x==0):
    middle = int((low + high)/2)
    if fruits[middle] == searchFruit:
        x = middle
    elif fruits[middle] < searchFruit:
        low = middle + 1
    else:
        high = middle - 1

print(x)
```

- Insertion Sort
  - Items from the input array are copied one at a time to the output array
  - Each new item is inserted into the right place so that the output array is always in order
  - Considerably faster than the bubble sort for a smaller number of data items
  - Iterative process

```
def insertionSort(array):
    for i in range(1, len(array)):

        currentValue = array[i]
        position = i

        while position > 0 and array[position-1] > currentValue:
            array[position] = array[position-1]
            position= position - 1

        array[position] = currentValue
    return (array)

array = [6,2,9,7,15]
insertionSort(array)
print(array)
```

- Bubble Sort
  - The list is divided into two sublists: sorted and unsorted.
  - The largest element is bubbled from the unsorted list and moved to the sorted sublist.
  - After that, the wall moves one element back, increasing the number of sorted elements and decreasing the number of unsorted ones.
  - Each time an element moves from the unsorted part to the sorted part one sort pass is completed.
  - Given a list of n elements, bubble sort requires up to n-1 passes (maximum passes) to sort the data.

```
array = [4,2,5,1,6,7,8,3]
for i in range(len(array)):
    for j in range(len(array)-1):
        if array[j] > array[j+1]:
            temp = array[j]
            array[j] = array[j+1]
            array[j+1] = temp
            print(array)
```

- The performance of either sort routine is the best when the data is already in order and there are a small number of data items.

- ***Linked Lists***:
  - Can be represented as two 1-D arrays -string array for data values and integer array for pointer values
  - Creating a Linked list →Setting values of pointers in free list and empty data linked list

```
FOR Index ← 1 TO 49
    NameList[Index].Pointer ← Index + 1
ENDFOR
NameList[50].Pointer ← 0
HeadPointer ← 0
FreePointer ← 1
```

A user-defined record type should first be created to represent a node's data and pointer:

```
Structure ListNode
    Dim Name As String
    Dim Pointer As Integer
End Structure
```

- Inserting into a Linked List

```
Procedure LinkedListInsertion (NewItem)
'place item at first free node
If FreePointer <> null Then
    ArrayLinkedList (FreePointer).Name ← NewItem
    'keep track of next free node in free list
    NextFreeNodeAddress ← ArrayLinkedList (FreePointer).Pointer
    'use this pointer variable to go through each node
    CurrentPointer ← HeadPointer
    'search for position where to insert item
    While (ArrayLinkedList (CurrentPointer).Name < NewItem
    and(CurrentPointer <> null)
        PreviousPointer ← CurrentPointer
        CurrentPointer ← ArrayLinkedList (CurrentPointer).Pointer
    End While
    'if node to be inserted at start of linked list with or without nodes
    If CurrentPointer = HeadPointer Then
        ArrayLinkedList(FreePointer).Pointer ← HeadPointer
        HeadPointer ← FreePointer
    Else
        'if node to be inserted between existing or after all nodes
        ArrayLinkedList (FreePointer).Pointer ← CurrentPointer
        ArrayLinkedList(PreviousPointer).Pointer ← FreePointer
    End IF
        'set freepointer to point to new free node
        FreePointer ← NextFreeNodeAddress
Else
    Output "No free space available"
End If
```

```
PROCEDURE AddItem(NewItem)
//
NameList[FreePointer].Name ← NewItem
CurrentPointer ← HeadPointer
//
REPEAT
IF NameList[CurrentPointer].Name < NewItem
THEN
PreviousPointer ← CurrentPointer
CurrentPointer ← NameList[CurrentPointer].Pointer
ENDIF
UNTIL NameList[CurrentPointer].Name > NewItem
//
IF CurrentPointer = HeadPointer
THEN
NameList[FreePointer].Pointer ← HeadPointer
HeadPointer ← FreePointer
ELSE
NameList[FreePointer].Pointer
← NameList[PreviousPointer].Pointer
NameList[PreviousPointer] ← FreePointer
ENDIF
FreePointer ← NameList[FreePointer].Pointer
ENDPROCEDURE
```

-Input NewItem

-Store NewItem in next free space

-Set Current to value at Start

-Read values in list following pointers.

-Until Current value in list > NewItem

-Pointer of Previous points to NewItem

-NewItem points to Current

-Update free space list

-Mention of any special cases e.g. NewItem being First in list // list empty // list full // no free space

- Searching a Linked List

```
Procedure SearchItem(NewItem)
    found ← false
    ' use currentpointer to go through each node
    CurrentPointer ← HeadPointer
    'search until found or end of linked list
    While CurrentPointer <> 0 and found = false
        ' if there is a match
        If ArrayLinkedList(CurrentPointer).Name = NewItem Then
            Print "item found at address" & CurrentPointer
            Set found to true
        End If
        ' go to next node
        CurrentPointer ← ArrayLinkedList(CurrentPointer).Pointer
    End While
    If found = false Then
        Print "Item not found in linked list"
    End If
```

- Deleting an Item from a Linked List
  1. Use a Boolean value to know when an item has been found and deleted (initially false)
  2. Use a pointer (CurrentPointer) to go through each node's address
  3. If new item is found at the header:
     a. Set head pointer to pointer of node at CurrentPointer
     b. Set pointer od node at CurrentPointer to free pointer
     c. Free pointer points to CurrentPointer
     d. Set Boolean value to True
  4. Otherwise:
     a. Search for Item while end of linked list not reached and Boolean value is false
        i. Use a Previous Pointer to keep track of the node located just before the one deleted
        ii. CurrentPointer point's to next node's address
        iii. If data in node at CurrentPointer matches SearchItem
           - Set pointer of node at PreviousPointer to pointer of node at CurrentPointer
           - Set pointer of node at CurrentPointer to FreePointer
           - Set FreePointer to CurrentPointer
           - Boolean value becomes true
  5. If Boolean value is false
     a. Inform user that item to be deleted has not been found

```
Procedure DeleteItem (Item)
    found ← false
    CurrentPointer ← HeadPointer
    ' if item to be deleted is at first node itself
    If ArrayLinkedList(CurrentPointer).Name= Item Then
        HeadPointer ← ArrayLinkedList(CurrentPointer).Pointer
        ArrayLinkedList(CurrentPointer).Pointer ← FreePointer
        FreePointer ← CurrentPointer
        found ← true
    Else
        ' if item to be deleted found after first node
        While CurrentPointer <> 0 And found = false
            PreviousPointer ← CurrentPointer
            ' go to next node
            CurrentPointer←ArrayLinkedList(CurrentPointer).Pointer
            ' if there is a match
            If ArrayLinkedList(CurrentPointer).Name = Item Then
                ArrayLinkedList(PreviousPointer).Pointer ←
                ArrayLinkedList(CurrentPointer).Pointer
                ArrayLinkedList(CurrentPointer).Pointer ← Free
                Pointer
                FreePointer ← CurrentPointer
                ' item was found and deleted
                found ← true
            End If
        End While
    End If
    If found = False Then
        Print "item to be deleted has not been found"
    End If
End Procedure
```

- ***Stacks***:
  - Stack – an ADT where items can be popped or pushed from the top of the stack only
  - LIFO – Last In First Out data structure

POPPING

```
PROCEDURE PopFromStack
    IF TopOfStack = -1
        THEN
            OUTPUT "Stack is already empty"
        ELSE
            OUTPUT MyStack[ TopOfStack ] "is popped"
            TopOfStack ← TopOfStack – 1
    ENDIF
ENDPROCEDURE
```

PUSHING

```
PROCEDURE PushToStack
    IF TopOfStack = MaxStackSize
        THEN
            OUTPUT "Stack is full"
        ELSE
            TopOfStack = TopOfStack + 1
            MyStack[TopOfStack] = NewItem
    ENDIF
ENDPROCEDURE
```

Use of Stacks:

- **Interrupt Handling**
  - The contents of the register and the PC are saved and put on the stack when the interrupt is detected
  - The return addresses are saved onto the stack as well
  - Retrieve the return addresses and restore the register contents from the stack once the interrupt has been serviced
- **Evaluating mathematical expressions held in Reverse Polish Notation**
- **Procedure Calling**
  - Every time a new call is made, the return address must be stored
  - Return addresses are recalled in the order 'last one stored will be the first to be recalled'
  - If too many nested calls then stack overflow

- ***Queues:***
  - Queue – an ADT where new elements are added at the end of the queue, and elements leave from the start of the queue
  - FIFO – First In First Out Data structure

  - **Creating a Circular Queue:**

```
PROCEDURE  Initialise
     Front = 1
     Rear = 6
     NumberInQueue := 0
END PROCEDURE
```

  - **To add an Element to the Queue:**

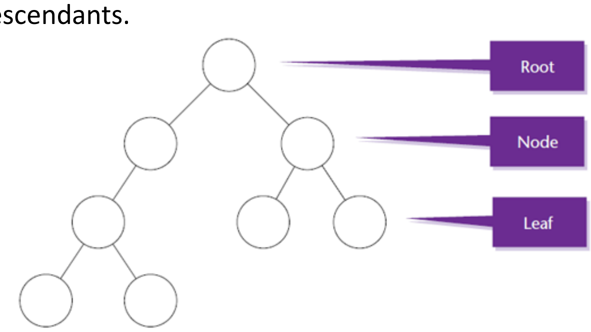
```
PROCEDURE EnQueue
     IF NumberInQueue == 6
        THEN Write ("Queue overflow")
     ELSE
        IF Rear == 6
           THEN Rear = 1
           ELSE Rear = Rear + 1
        ENDIF
        Q[Rear] = NewItem
        NumberInQueue =NumberInQueue +1
     ENDIF
ENDPROCEDURE
```

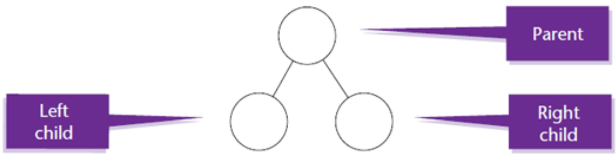
  - The front of the queue is accessed through the pointer Front.
  - To add an element to the queue, the pointers have to be followed until the node containing the pointer of 0 is reached → the end of the queue, and this pointer is then changed to point to the new node.
  - In some implementations, 2 pointers are kept: 1 to the front, and 1 to the rear. This saves having to traverse the whole queue when a new element is to be added.
  - To Remove an Item from the Queue

```
PROCEDURE DeQueue
     IF NumberInQueue == 0
          THEN Write ("Queue empty")
     ELSE
          NewItem = Q[Front]
          NumberInQueue =
NumberInQueue – 1
          IF Front ==6
               THEN Front = 1
          ELSE
               Front = Front + 1
          ENDIF
     ENDIF
END PROCEDURE
```

- Items may only be removed from the front of the list and added to the end of the list

- ***Binary Trees***:
  - Dynamic Data structure → can match the size of data requirement.
  - Takes memory from the heap as required and returns memory as required, following a node deletion
  - An ADT consisting of nodes arranged in a hierarchical fashion, starting with a root node
  - Usually implemented using three 1-D arrays
  - In a binary tree, a node can have no more than two descendants.

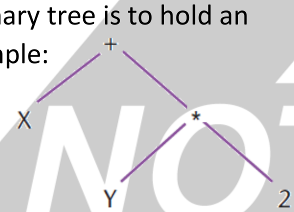




- A binary tree node is like a linked list node but with two pointers, LeftChild and RightChild.
- Binary trees can be used in many ways. One use is to hold an ordered set of data. In an ordered binary tree all items to the left of the root will have a smaller key than those on the right of the root. This applies equally to all the sub-trees.
- Tree algorithms are invariably recursive.
- To insert data into an ordered tree the following recursive algorithm can be used:

PROCEDURE insert(Tree, Item)
IF Tree is empty THEN create new tree with Item as the root.
ELSE IF Item < Root
THEN insert(Left sub-tree of Tree, Item)
ELSE insert(Right sub-tree of Tree, Item)
ENDIF
ENDIF
ENDPROCEDURE

- Another common use of a binary tree is to hold an algebraic expression, for example:
X + Y * 2
could be stored as:



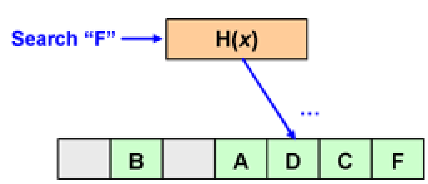
- ***Algorithm to search a Binary Tree:***

```
START at Root Node
REPEAT
        IF WantedItem = ThisItem
                THEN Found = TRUE
                ELSE
                        IF  WantedItem > ThisItem
                                THEN Follow Right Pointer
                                ELSE Follow Left Pointer
UNTIL Found or Null Pointer Encountered
```

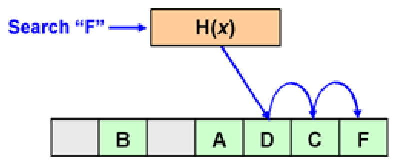
```
//binary tree search
INPUT SearchAnimal
IsFound ← FALSE
Current ← RootPtr
REPEAT
   IF SearchAnimal == Data[Current]
     THEN
        //found
        OUTPUT 'Found'
        IsFound ← TRUE
     ELSE
        IF SearchAnimal > Data[Current]
          THEN
            // move right
            Current ← RightPtr[Current]
          ELSE
            Current ← LeftPtr[Current]
        ENDIF
   ENDIF
UNTIL IsFound == True OR Current = 0

IF IsFound == False
  THEN
    OUTPUT SearchAnimal ' not found'
ENDIF
```
[5]

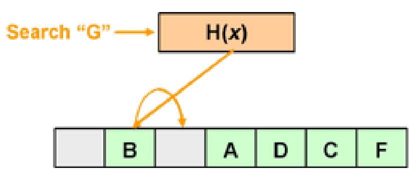
- ***Hash Tables***:
  - A hash table is a collection of items which are stored in such a way as to make it easy to find them later.
  - Each position of the hash table, often called a slot, can hold an item and is named by an integer value starting at 0.
  - Given some key, we can apply a hash function to it to find an index or position that we want to access.
  - To find data from the hash table, we need a key to search for. From this key, we can calculate the hash code. This tells us where in the data array we need to start searching.



- Because of the collision resolution of the add operation, the target data might reside at a location other than the element referred to by the hash code.
- Therefore, it is necessary to probe the hash table until an empty hash element is found, and for an exact match between each data item and the given key. (The probing stops at an empty element, since it signals the end of where potential data might have been stored.)



- Consider a situation where 'G' maps to the same hash code as 'B', and a search is undertaken. The retrieval algorithm will start looking at data items starting at that hash code, and continue comparing each hash item's contents for a match with 'G', until either the blank element is found, or (if the array is full) the probing loops back and ends up where the traversal started.



- ***Search Algorithm for a Hash Table***:
  - Calculate the hash code for the given search key
  - Access the hash element
  - If the hash element is empty, the search has immediately failed.
  - Otherwise, check for a match between the search and data key
  - If there is a match, return the data.
  - If there is no match, probe the table until either:
  - An match is found between the search and data key
  - A completely empty hash element is found.

- We must weigh the trade-offs between an algorithm's time requirement and its memory requirements.
  - For example, an array-based list search function is O(1), but a linked-list-based list search function is O(n).
  - Search for Items in Arrays is much faster, but insert and delete operations are much easier on a linked-list-based list implementation.
  - However, linked lists require more memory
  - When selecting an ADT's implementations, we must consider how frequently particular ADT operations occur in a given application.
  - If the problem size is always small, we can probably ignore an algorithm's efficiency → use the simplest algorithm
  - Order-of-magnitude(O(x)) analysis focuses on large problems.

4.1.3 Abstract Data Types (ADTs)

- A collection of data and a set of operations on those data
  - Stack
  - Queue
  - Linked list
  - Dictionary / Hash Table
  - Binary tree
  - Algorithms for the ADTs above has been shown in Section 4.1.2
  - Many of the ADTs described are "dynamic" → can change in size during run time, taking up more or less memory as required
  - Data structures not available as built-in types in a programming language need to be constructed from those available data structures which are built-in the language.
  - For example, a linked list is to be implemented using these array data structures

Define a record type, ListNode, for each node:

```
TYPE ListNode
        DECLARE Pointer : INTEGER
        DECLARE Name : STRING
ENDTYPE
```

- Implementation of different ADTs:
  - Using built-in data types to create an ARRAY
  - Using classes within subclasses in OOP

4.1.3 Recursion

- Allows us to define a function that calls itself to solve a problem by breaking it into simpler cases.
  - Important technique used in imperative and declarative programming
  - Uses a stack to store return addresses when compiled
  - When a function is defined in terms of itself
  - Breaks down a problem into smaller pieces which you either already know the answer to, or can solve by applying the same algorithm to each piece, and then combining the results.
- The essential features of a recursive process:
  1. A stopping condition which when met, means that the routine will not call itself and will start to "unwind"
  2. For input values other than the stopping condition, the routine must call itself
  3. The stopping condition must be reached after a finite number of calls → base case

1. Infinite recursion – when a function that calls itself recursively without ever reaching any base case causes a stack overflow, runtime error.

Advantages	Disadvantages
▪ Can produce simpler, more natural solutions to a problem	▪ Less efficient in terms of computer time and storage space ▪ A lot more storage space is used to store return addresses and states ▪ Could lead to infinite recursion

4.2 Algorithm Design Methods

4.2.1 Decision Tables

- Purpose → Determine logical conditions and consequential actions.
  - Decision tables are compact and precise ways of modelling complicated logic, such as that which you might use in a computer program.
  - They do this by mapping the different states of a program to an action that a program should perform.
  - Decision tables take on the following format:

The four quadrants

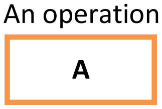
Conditions	Condition alternatives
Actions	Action entries

  - The limited-entry decision table is the simplest to describe. The condition alternatives are simple Boolean values, and the action entries are check-marks, representing which of the actions in each column are to be performed.
  - A technical support company writes a decision table to diagnose printer problems based upon symptoms described to them over the phone from their clients. They type the following data into the advice program:
    1. Printer does print
    2. Red light is flashing
    3. Printer is recognized
  - The program then uses the decision table to find the correct actions to perform, namely that of Check / Replace ink.

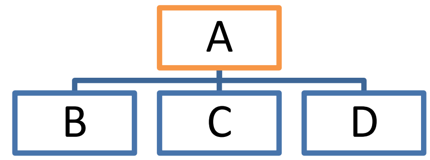
Printer troubleshooter									
		Rules							
Conditions	Printer does not print	Y	Y	Y	Y	N	N	N	N
	A red light is flashing	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognised	Y	N	Y	N	Y	N	Y	N
Actions	Check the power cable			X					
	Check the printer-computer cable	X		X					
	Ensure printer software is installed	X		X		X		X	
	Check/replace ink	X	X			X	X		
	Check for paper jam		X		X				

4.2.1 Jackson Structured Programming

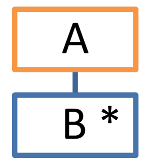
- JSP is a method for structured programming based on correspondences between data stream structure and program structure.
- JSP structures programs and data in terms of sequences, iterations and selections, and as a consequence it is applied when designing a program's detailed control structure, below the level where object-oriented methods become important
- An operation:



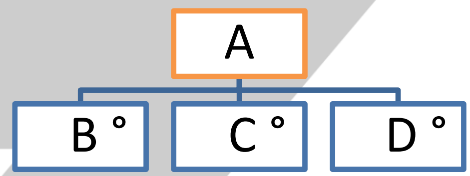
- A sequence of operations:



- An iteration:



- A selection:



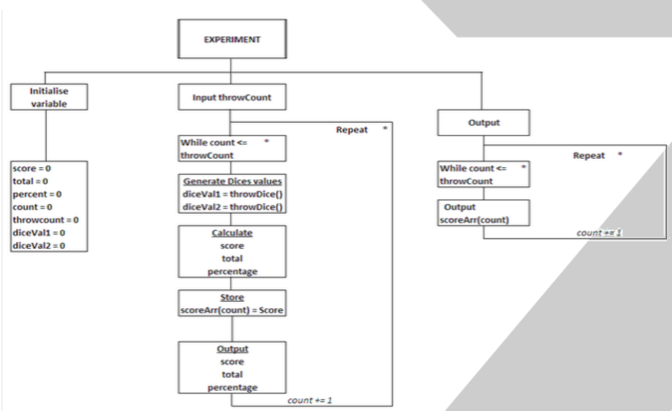
- JSP is more simplistic compared to a flowchart
- In a mathematical experiment, two six-sided dice, each labelled 1, 2, 3, 4, 5 and 6, are thrown a number of times. Each time they are thrown, the numbers on the two dice are added together. At the end of the experiment, a report is made of the number of times each score, from 2 to 12, has occurred. Additionally, the results are reported as a percentage of the total number of throws.This experiment is to be simulated by using a computer. The number of throws is to be set by the operator.

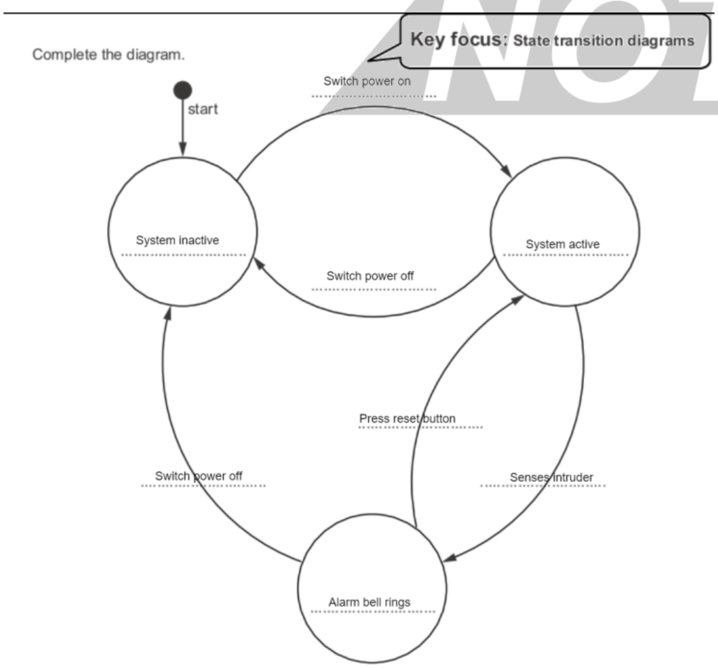
(a) Draw a Jackson diagram to illustrate how the problem may be broken down. [9]

○ Answer in Pseudocode:

Initialise
set totals to zero
generate two numbers
keep running total of throws
process two numbers
add one to total occurrences of score
calculate total score
add one to total occurrences of score
output totals of each total score
output percentages for each total
add one to total occurrences of score

○ Answer in JSP:





4.2.3 State Transition Diagrams

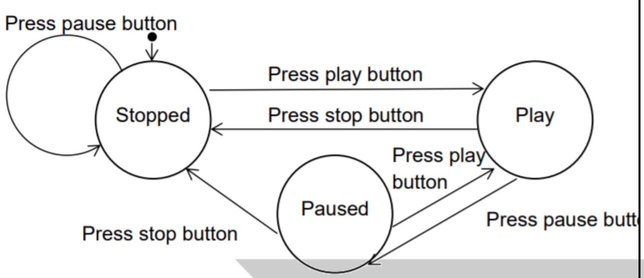
- State-transition diagrams are suitable for systems that operate as finite-state machines – these are systems that have a fixed number of different states that may change on an event or input.
- State transition diagrams give a visual representation of all the states that a system can have, the events such as inputs or timers that may result in transition between states, and the transitions between states.
- They may also show the conditions needed for an event(s) to cause a transition to occur (the guard condition), and the outputs or actions carried out as the result of a transition.
- There are different conventions for state-transition diagrams, but states are normally represented as nodes, transitions as interconnecting arrows, and events as labels on the arrows.
- Conditions are normally specified in square brackets after the event
- The initial state is indicated by an arrow with a black dot.
- Task 3 Paper 4 Pre-release 2015 p42

  An intruder detection system is inactive when the power is switched off. The system is activated when the power is switched on. When the system senses an intruder the alarm bell rings. A reset button is pressed to turn the alarm bell off and return the system to the active state.

  The transition from one state to another is as shown in the state transition table below.

Current state	Event	Next state
System inactive	Switch power on	System active
System active	Senses intruder	Alarm bell rings
System active	Switch power off	System inactive
Alarm bell rings	Press reset button	System active
Alarm bell rings	Switch power off	System inactive

The example below shows a simple state-transition diagram for a media player with three buttons: stop, play and pause. The initial state of the player is stopped. In each state, only the buttons for the other states can be pressed (e.g. in play, only the stop and pause buttons can be pressed).
Pressing the pause button when the player is stopped does not result in any change to the player.



The event (press pause when state is Stopped) that does not cause any change in state is indicated by the circular arrow. A finite-state machine can also be represented by a state-transition table, which lists all the states, all possible events, and the resulting state.

The following is the state-transition table for the diagram above:

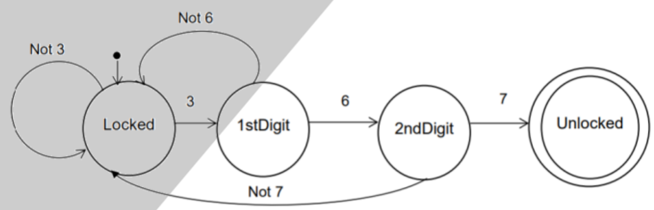
Current state	Event	Next state
Stopped	Press play button	Play
Stopped	Press pause button	Stopped
Play	Press stop button	Stopped
Play	Press pause button	Paused
Paused	Press play button	Play
Paused	Press stop button	Stopped

State-transition diagrams are also useful for showing the working of algorithms that involve a finite number of states. The following algorithm is for a three-digit combination lock where the correct combination to unlock is '367'. The initial state is Locked, each correct digit changes the state, until the combination unlocks the lock. An incorrect digit returns the lock to the original locked state.

```
DECLARE State : String
DECLARE Number : Integer

State ← Locked
INPUT Number
CASE OF Number
   3 : IF State = Locked
          THEN State ← 1stDigit
       ENDIF
   6 : IF State = 1stDigit
          THEN State ← 2ndDigit
          ELSE State ← Locked
       ENDIF
   7 : IF State = 2ndDigit
          THEN State ← Unlocked
          ELSE State ← Locked
       ENDIF
ENDCASE
```

A state-transition diagram for the algorithm is shown below:



The double line around the Unlocked state indicates that lock halts in this state – this is also known as the 'accepting state'.

Word/phrase	Meaning
Accepting state	A state the system reaches when the input string is valid
Event	Something that can happen within a system, such as a timer event, or an input to the system, that may trigger a transition to another state
Finite state machine (FSM)	A system that consists of a fixed set of possible states with a set of allowable inputs that may change the state and a set of possible outputs
Guard condition	A condition which must be met for a transition to occur from one state to another

State	The value or the position in which a system is at a given point
State transition diagram	A graphical representation of a finite state machine
State transition table	A table that shows all the states of an FSM, all possible inputs and the state resulting from each input
Transition	The change from one state to another state

4.3 FURTHER PROGRAMMING

4.3.1 Programming Paradigms

- A programming paradigm defines the style or model followed when programming.
  - Low-level programming
    - Machine code (binary – lowest level) or Assembly language
    - "Low" refers to the small/non-existent amount of abstraction between the language and machine language
    - Instructions can be converted to machine code without a compiler or interpreter
    - The resulting code runs directly on the specific computer processor, with a small memory footprint
    - Programs written in low-level languages tend to be relatively non-portable – code written for a Windows processor might not work on a Mac processor
    - Simple language, but considered difficult to use, due to numerous technical details that the programmer must remember.
  - Imperative programming
    - Uses a sequence of statements to determine how to reach a certain goal. These statements are said to change the state of the program as each one is executed in turn.

```
total = 0
number1 = 5
number2 = 10
number3 = 15
total = number1 + number2 + number3
```

    - Each statement changes the state of the program, from assigning values to each variable to the final addition of those values. Using a sequence of five statements the program is explicitly told how to add the numbers 5, 10 and 15 together.
  - Object-Oriented Programming
    - An extension of imperative programming.
      The focus is on grouping functions and data into logical classes, and instances of classes called objects.
  - Declarative Programming
    - Non-procedural and very high level (4th generation)
    - Control flow is implicit, not explicit like Imperative Programming
    - Programmer states only what the needs to be done and what the result should look like, not how to obtain it.
    - An important feature → backtracking – where a search goes partially back on itself, if it fails to find a complete match the first-time round
    - Goal – a statement we are trying to prove either true or false, effectively forms a query
    - Instantiation – giving a value to a variable in a statement

4.3.2 File Processing

- Records are user-defined data structures
  Defining a record structure for a Customer record with relevant fields (e.g. customer ID) in Python:

```
class CustomerRecord :
    def __init__(self) :
        self.CustomerID = 0
        self.CustomerName = ''
        self.TelNumber = ''
        self.TotalOrders = 0
```

- Files are needed to import contents (from a file) saved in secondary memory into the program, or to save the output of a program (in a file) into secondary memory, so that it is available for future use

Pseudocode:

- Opening a file:

    OPENFILE <filename> FOR READ/WRITE/APPEND

- Reading a file:

    READFILE <filename>

- Writing a line of text to the file:

    WRITEFILE <filename>, <string>

- Closing a file:

    CLOSEFILE

- Testing for end of the file:

    EOF()

Python:

- Opening a file

    variable = open("filename", "mode")

    Where the mode can be:

Mode	Description
r	Opens file for reading only. Pointer placed at the beginning of the file.
w	Opens a file for writing only. Overwrites file if file exists or creates new file if it doesn't
a	Opens a file for appending. Pointer at end of file if it exists or creates a new file if not

- Reading a file:
    - o Read all characters

        variable.read()

    - o Read each line and store as list

        variable.readlines()

- Writing to a file:
    - o Write a fixed a sequence of characters to file

        variable.write("Text")

    - o Write a list of string to file

        variable.write["line1", "line2", "line3"]

- Using a direct access or Random File allows us to read records directly. The term 'random' is misleading since records are still read from and written to the file in a systematic way.

Pseudocode:

- Opening a file, using the RANDOM file mode, where once the file has been opened, we can read and write as many times as we would like, in the same session:

    OPENFILE <filename> FOR RANDOM

- Move a pointer to the disk address for the record before reading/writing to a file can occur:

    SEEK <filename>, <address>

Each record is given an 'address' at which it is to be written – the record key.

- Write a record to the file:

    PUTRECORD <filename>, <identifier>

- Read a record from a file:

    GETRECORD <filename>, <identifier>

- Close the file:

    CLOSE <filename>

Algorithms for File Processing Operations for Serial and Sequential Files:

- Display all records:

File mode: READ

```
Loop
    Read a new record
    Output data
Until the end of the file is reached
```

- Search for a record:

File mode: READ

```
Input the name to find
Loop
    Read the next record
    IF matching record, then  Output data
Until 'Found' or the end of the file is reached
```

*Special Case: If the records in a sequential file are of a fixed length, a record can be retrieved using its relative position in the file. So the start position in the file could be calculated for the record with the key number 15 for example. *

- Add a new record – Serial Organisation:

File mode: APPEND

```
Input the new data
Append the new record to the end of the file
```

- Add a new record – Sequential Organisation:

**Some file processing tasks, like this one, require the use of two files, because serial/sequential files can only be opened for either reading from or writing to in the same session. **

This requires the use of two files: the original FileA and a new FileB.

```
Input the name to add
Search FileA
Loop
    Read record from FileA
    If current record > insert name
        Write new record to FileB
    Write current record to FileB
Until position to insert is found
Write all remaining records in FileA
    to FileB
Delete FileA
Rename FileB as FileA
```

- Delete a record:

This requires the use of two files: the original FileA and a new FileB.

```
Input the name to delete
Search FileA
Loop
    Read record from FileA
    If the current record is not the one to delete
        Write current record to FileB
Until EOF(FileA)
Delete FileA
Rename FileB as FileA
```

- Amend an existing record:

This is similar to the delete algorithm. It requires the use of two files: the original FileA and a new FileB.

```
Input the name to amend and the new data values
Search FileA
Loop
    Read record from FileA
    If the current record is not the one to amend
        Write this record to FileB
Until found
When the record is found write the new data to FileB
Write all remaining records from FileA to FileB
Delete FileA
Rename FileB as FileA
```

Python example of Sequential File Handling:

```
class CarRecord :
 def __init__(self) :
     self.VehicleID = "dummy"
     self.Registration = ""
     self.DateOfRegistration = date(1990,1,1)
     self.EngineSize = 0
     self.PurchasePrice = 0.0

def SaveData(Car) :
 # file channel for car records
 CarFile = open('CarFile.DAT','wb')
 for i in range(100): # loop for each array element
 # write a whole record to the binary file
     pickle.dump(Car[i], CarFile)
 CarFile.close() # close file

def LoadData() :
 CarFile = open('CarFile.DAT','rb') # open file for binary read
 Car = [] # start with empty list
 EoF = False
 while not EoF : # check for end of file
 try :
     Car.append(pickle.load(CarFile)) # append record from file to end of list
 except :
     EoF = True
 CarFile.close()
 return Car

def OutputRecords(Car) :
 for i in range(100): # loop for each array element
     print(Car[i].VehicleID) # write one field

def main() :
 ThisCar = CarRecord()
 #Car =[ThisCar for i in range(100)] # only run this 1st time
 #SaveData(Car) # only run this first time
 Car = LoadData() # from existing file
 OutputRecords(Car)
 # add more records
 i = int(input('Record Number? '))
 while i != 0 :
     Car[i].VehicleID = input('Vehicle ID: ')
     Car[i].Registration = input('Registration: ')
     Car[i].DateOfregistration = (input('Registration
    Date: '));
     Car[i].EngineSize = int(input('Engine size: '))
     Car[i].PurchasePrice = float(input('Purchase
    price: '))
     i = int(input('next Record Number? '))
 OutputRecords(Car)
 SaveData(Car)

main()
```

Algorithms for File Processing Operations for Random Files:

• Display all records:

File mode: RANDOM

```
Loop for every key number
    Read the record
    Output record data
```

• Add a new record:

File mode: RANDOM

If the key number is hashed, generate the key number. Check this key number has not already been used before using SEEK then PUTRECORD to write the data.

If duplicate keys are possible there will need to be a strategy for dealing this this.

Python:

```
def AddRecord(CustomerData, Customer):
    Address = Hash(Customer.CustomerID)
    while CustomerData(Address).CustomerID != 0 :
        Address += 1
        if Address = 1000 :
            Address = 0
    CustomerData[Address] = Customer
```

• Delete a record:

File mode: RANDOM

Strategy 1: SEEK to find the record and PUTRECORD to overwrite the data items with the dummy values. A following SEEK and GETRECORD will recognise this record no longer exists.

Strategy 2: Include an extra field in the original record structure to act as a Boolean flag. When a record is deleted, the flag value is changed to indicate 'deleted record'. We are effectively amending the record to mark it as deleted:

1 SEEK and GETRECORD to establish we have the correct record.

2 Change the flag value.

3 SEEK (with the same key number) then PUTRECORD to write the amended data.

• Amend an existing record:

File mode: RANDOM

1 SEEK and GETRECORD to establish we have the correct record.

2 Input the new data values.

3 SEEK (with the same key number) then PUTRECORD to write the amended data.

• Search for a record:

File mode: RANDOM

Case 1 If the key number is known, read the data – SEEK then GETRECORD – and output the data.

Case 2 If the key number has been hashed from the data, then a hash of the required data will give the key number.

Case 3 The worst case is to loop through the records with each key number in turn, until it is found.

Python:

```
def FindRecord(CustomerData, ID):
    Address = Hash(ID)
    while CustomerData[Address].CustomerID != ID :
        Address += 1
        if Address = 1000 :
            Address = 0
    return (Address)
```

Python example of Random File Handling:

```
RECORDSIZE = 50 # 20 + 10 + 8 + 4 + 8

class CarRecord :
 def __init__(self) :
     VehicleID = "dummy"
     VehicleID = VehicleID.ljust(20)
     self.VehicleID = VehicleID.encode('utf-8')
     Registration = " "
     Registration = Registration.ljust(10)
     self.Registration = Registration.encode('utf8')
     self.DateOfRegistration = date(1990,1,1)
     self.EngineSize = 0
     self.PurchasePrice = 0.0

def InitialiseFile() :
 CarFile = open('CarFile.DAT','wb') # file for car records
 for i in range(100): # loop for each array element
     Address = i * RECORDSIZE + 1
     CarFile.seek(Address, 0)
     # write a whole record to the binary file
     pickle.dump(CarRecord(), CarFile)
 CarFile.close() # close file

def InputNewRecordData() :
 ThisCar = CarRecord()
 VehicleID = input('Vehicle ID: ')
 VehicleID = VehicleID.ljust(20)
 ThisCar.VehicleID = VehicleID.encode('utf-8')
 Registration = input('Registration: ')
 Registration = Registration.ljust(10)
 ThisCar.Registration = Registration.encode('utf8')
 ThisCar.DateOfregistration = (input('Registration Date: '));
 ThisCar.EngineSize = int(input('Engine size: '))
 ThisCar.PurchasePrice = float(input('Purchase price: '))
 return ThisCar

def Hash(reg) :
 result = ord(reg[0]) * RECORDSIZE + 1
 print('Hashed to ',result)
 return result

def SaveToFile(ThisCar, CarFile) :
 Address = Hash(ThisCar.Registration.decode('utf8'))
 CarFile.seek(Address, 0)
 pickle.dump(ThisCar, CarFile)# write a whole record to the binary file

def OpenFileForUpdate() :
 CarFile = open('CarFile.DAT','rb+') # open file for update
 return CarFile

def FindRecord(reg, CarFile) :
 Address = Hash(reg)
 CarFile.seek(Address, 0)
 ThisCar = pickle.load(CarFile) # load record from file
 return ThisCar

def OutputData(ThisCar) :
 print(ThisCar.VehicleID) # write one field

def main() :
 InitialiseFile() # only run this procedure the first time
 CarFile = OpenFileForUpdate()
 ThisCar = CarRecord()
 # add records
 Answer = input('add a record? (Y/N) ')
 while Answer != 'N' :
     ThisCar = CarRecord()
     ThisCar = InputNewRecordData()
     SaveToFile(ThisCar, CarFile)
     Answer = input('add a record? (Y/N) ')
 # find records
 Answer = input('find a record? (Y/N) ')
 while Answer != 'N' :
     Reg = input('Give vehicle registration: ')
     ThisCar = FindRecord(Reg, CarFile)
     OutputData(ThisCar)
     Answer = input('find a record? (Y/N) ')
 CarFile.close()
```

4.3.3 Exception Handling

- An exception is a runtime error/ fatal error / situation which causes a program to terminate/crash
- Exception-handling – code which is called when a run-time error or "exception" occurs to avoid the program from crashing
- When an exception occurs, we say that it has been "raised." You can "handle" the exception that has been raised by using a try block.
- A corresponding except block "catches" the exception and prints a message back to the user if an exception occurs.

e.g.

```
EoF = False
while not EoF :  # check for end of file
    try :
        Car.append(pickle.load(CarFile))
    except :
        EoF = True
```

4.3.4 Use of development tools / programming environments

- Integrated Development Environment → an application that provides several tools for software development. An IDE usually includes: source code editor, debugger and automated builder
- Features in editors that benefit programming:
  - Syntax Highlighting – keywords are coloured differently according to their category
  - Automatic indentation – after colons for example to make code blocks more distinct allowing for better code readability
  - A library of preprogrammed subroutines that can be implemented into a new program to speed up the development process

Compiler	Interpreter
translates source code (e.g. Python code) into machine code which can be run executed by the computer	directly executes/performs instructions written in a programming language by translating one statement at a time
It takes large amount of time to analyze the source code but the overall execution time is comparatively faster.	It takes less amount of time to analyze the source code but the overall execution time is slower.
Generates intermediate object code which further requires linking, hence requires more memory.	No intermediate object code is generated, hence are memory efficient.
It generates the error message only after scanning the whole program. Hence debugging is comparatively hard.	Continues translating the program until the first error is met, in which case it stops. Hence debugging is easy.
Programming language like C, C++ use compilers.	Programming language like Python, Ruby use interpreters.

- Systems that require high performance and for the long run should be written in compiled languages like C, C++
- Systems that need to be created quickly and easily should be written in interpreted languages

Features available in debuggers:

- **Stepping -** traces through each line of code and steps into procedures. Allows you to view the effect of each statement on variables
- **Breakpoints -** set within code; program stops temporarily to check that it is operating correctly up to that point
- **Go to File/Line -** Look on the current line. with the cursor, and the line above for a filename and line number. If found, open the file if not already open, and show the line. Use this to view source lines referenced in an exception traceback and lines found by Find in Files. Also available in the context menu of the Shell window and Output windows.
- **Debugger (toggle) -** When active, code entered in the Shell or run from an Editor will run under the debugger. In the Editor, breakpoints can be set with the context menu. This feature is still incomplete and somewhat experimental.
- **Stack Viewer -** Show the stack traceback of the last exception in a tree widget, with access to local and global variables.
- **Auto-open Stack Viewer** - Toggle automatically opening the stack viewer on an unhandled exception.

4.4. Software Development

4.4.1 Software Development Processes

- Program Generator – a program that writes source-code programs directed by a series of parameters/rules enabling an individual to create a program with less times, effort and programming knowledge
- Program Library – a collection of prewritten code that can be reused as needed to develop programs to speed up the development process e.g. using the Random function from the Math Class to generate random numbers, the Math Class is a component of the Python Library

4.4.2 Testing

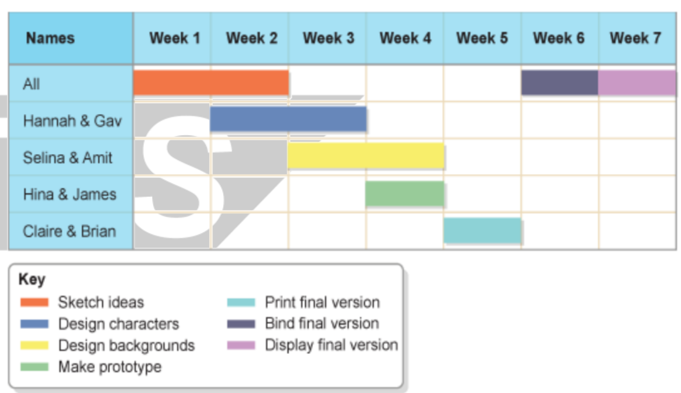
- Programs are written by humans, and so errors are bound to occur, so regular **testing** is crucial to ensuring a program is resilient under various circumstances
- Types of Errors:
  - Syntax – incorrect use of programming language, detected by the compiler/interpreter e.g. typos, missing a colon ':'
  - Logical – error in the programmer's logic e.g. multiplying two numbers instead of adding them
  - Runtime – error that is detected on when the program runs and causes the program to crash e.g. division by 0
- Test Plans - list of requirements designed to ensure that the coded solution works as expected. The test plan will include specific instructions about the data and conditions the program will be tested with.
- Testing strategies:
  - **Dry run –** Working through and algorithm or program code with test data, recording the variable values in a trace table as they change

- **Walkthrough**
  - It is not a formal process/review
  - It is led by the programmers
  - Programmer guides the participants through the document according to his or her thought process to achieve a common understanding and to gather feedback.
  - Useful for the people if they are not from the software discipline, who are not used to or cannot easily understand software development process.
- **White-box**
  - Testers examine each line of code for the correct logic and accuracy
- **Black-box**
  - Programmer uses test data, for which the results are known, and compares the results from the program with those expected
  - The testing only considers the inputs and outputs produced
  - Code is viewed as being inside a black-box
- **Integration**
  - Performed when two or more tested units are combined into a larger structure. The test is often done on both the interfaces between the components and the larger structure being constructed, if its quality property cannot be assessed from its components.
- **Unit Testing** is done at the lowest level.
  - Tests the basic unit of software, which is the smallest testable piece of software, and is often called "unit", "module", or "component" interchangeably.
- **Alpha**
  - Done within the software company
  - Program may still be incomplete
  - Employers not involved in the programming may find errors missed by the programmer
- **Beta**
  - Follows alpha testing
  - Software is made available to a few selected testers
  - Program is virtually complete
  - Testers provide constructive criticism
- **Acceptance**
  - Done by the client
  - Errors discovered when program runs on client's hardware and OS
  - Software is complete, and the developer must prove to the client that the software meets all the requirements

- Types of Test Data:
  - Normal – within acceptable range and follows rules
  - Borderline – at the limits of the range set
  - Invalid – completely out of range and doesn't follow any rules, should be rejected

4.4.3 Project Management

- Think of Microsoft, over a million lines of code wasn't written by just one person. The bigger picture is broken down into modules and split amongst teams of people.
- With such large teams, keeping everyone on track is crucial to achieving the goal and hence a Project Manager is needed to direct the breakdown and processes of development.
- Project planning techniques include the use of GANTT and PERT charts.
- GANNT chart – a horizontal bar chart of tasks with clear start and ends end dates, named after Henry L. Gantt in 1917

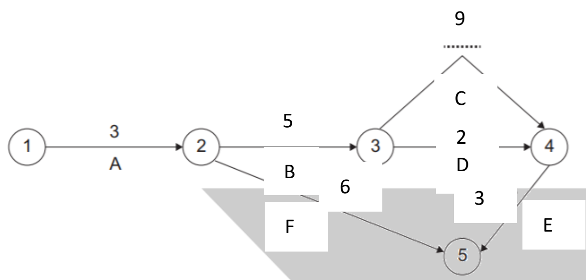


- PERT chart – Program Evaluation Review Technique charts is a network model that allows for the randomness in activity completion times. It follows the Critical Path Method to use a fixed time estimate for each activity within a project/program

A software development project consists, in part, of these activities.

		Weeks to complete
A	identify requirements	3
B	produce design	5
C	write code	9
D	black box testing	2
E	acceptance testing	3
F	prepare documentation	6

From this data, a Program Evaluation Review Technique (PERT) chart is constructed.



(a) Complete the PERT chart.

(b) **(i)** State the critical path.

1,23,4,5

(ii) State the minimum time for the completion of this development.

20

(c) For activity D:

(i) state the earliest start time.

8

(ii) state the latest finish time.

17

CIE A2-LEVEL COMPUTER SCIENCE//9608

WWW.ZNOTES.ORG