1. (a)
    - Binary search can only be used on sorted array. On the other hand, linear search can be used on both sorted and unsorted array.
    - Binary search has a time complexity of $O(\log n)$ when linear search is $O(n)$.

   (b) Linear search is more appropriate here as it has $O(n)$ time complexity. If binary search is to be used here, we must first sort the array, which has $O(n \log n)$, on top of the $O(\log n)$ time complexity of the binary search itself.

   (c)
```
FUNCTION LinearSearch(A: Array[1:N] OF INTEGER, k:INTEGER)
RETURNS INTEGER
    val <- MAXINT
    pos <- -1
    FOR i <- 1 TO N:
    IF A[i] > k and A[i] < val:
        val <- L[i]
        pos <- i
    RETURN pos
ENDFUNCTION
```

   (d) The issue: Recursive function requires high memory and computing resources . Every recursive function pushes state data into the call stack memory, if the depth of the binary search exceeds the call stack memory, it will result in a run time stack overflow error.

   Choice for the application above: Iterative version. As there are no function calls and the all data state changes within a local function scope.

   (e) We have

```
FUNCTION Helper(A: ARRAY[1:N] OF INTEGER, k:  INTEGER, lb:
INTEGER, ub:INTEGER)


    IF lb > ub THEN
        RETURN -1
    ENDIF


    mid <- (lb + ub) DIV 2
    IF A[mid] = k THEN
        RETURN mid
    ENDIF
    IF k < A[mid] THEN
        RETURN Helper(A, k, lb, mid - 1)
    ELSE
        RETURN Helper(A, k, mid + 1, ub)
    ENDIF
ENDFUNCTION


FUNCTION BinSearch_1(A: ARRAY[1:N] OF INTEGER, k:  INTEGER )
    RETURNS INTEGER RETURN Helper(A, k, 1, N)
ENDFUNCTION
```

(f) We have

```
FUNCTION BinSearch_2(A: ARRAY[1:N] OF INTEGER, k:  INTEGER )
RETURNS INTEGER
    lb <- 1
    ub <- N
    ret <- -INF //smallest numerical value
    WHILE lb <= ub //
        mid <- (lb+ub) DIV 2
        IF A[mid] < k THEN //
            IF A[mid] > ret THEN
                ret = A[mid]
            ENDIF
            lb <- mid + 1
        ELSE
            ub <- mid - 1 ## final ret will be
        ENDIF
    IF ret > -INF THEN
        RETURN ret
    ELSE
        RETURN -1
    ENDIF
ENDFUNCTION
```

(g) We have

Category	Test Case	Expected Result
Valid	BinSearch_2([2,4,6,8], 5)	4
Boundary	BinSearch_2([2,4,6,8], 1)	-1
Boundary	BinSearch_2([2,4,6,8], 2)	-1
Boundary	BinSearch_2([2,4,6,8], 10)	8
Boundary	BinSearch_2([2,4,6,8], 8)	6
Invalid	BinSearch_2([2,4,6,8], "a")	Runtime Error

(h) We have

```
FUNCTION Helper(A:ARRAY[1:N] OF INTEGER, k:  INTEGER, lb:
INTEGER, ub:  INTEGER, flag:  STRING, index:  INTEGER) RETURNS
INTEGER
    IF lb > ub THEN // base case
        RETURN index
    ENDIF
    mid <- (lb+ub)DIV 2 // recursive step
    IF k = A[mid] THEN
        index <- mid // for key found
        IF flag = "L" THEN
            RETURN Helper(A, k, lb, mid-1, flag, index) // search
left sublist
        ELSE
            IF flag = "R" THEN // search right sublist
                RETURN Helper(A, k, mid+1, ub, flag, index)
            ENDIF
        ENDIF
    ELSE IF k < array[mid] THEN //
        RETURN Helper(A, k, lb, mid-1, flag, index)
    ELSE: //
        RETURN Helper(A, k, mid+1, ub, flag, index)
    ENDIF
ENDFUNCTION


FUNCTION BinSearch_3(A:ARRAY[1:N] OF INTEGER, k:  INTEGER)
RETURNS ARRAY[1:2] OF INTEGER //
DECLARE result:  ARRAY[1:2] OF INTEGER
    result[1] <- Helper(A, k, 1, N, "L", -1)
    result[2] <- Helper(A, k, 1, N, "R", -1)
    RETURN result
ENFUNCTION
```

2. (a)

Source	Directly Connected Neighbours
`Node(1)`	`[Node(2), Node(3)]`
`Node(2)`	`[Node(4)]`
`Node(3)`	`[Node(2)]`
`Node(4)`	`[Node(3)]`

(b) The methods are:

i. `append (n: Object): None` // add object `n` to the end of the `List`

ii. `getHead(): Object` // returns the first object in the `List`, `None` if empty `List`

iii. `getNext(n: Object): Object` // returns the next object from the `List` after `n` , returns `None` if `n` is the last node in the `List`

(c) Since the set of nodes in the graph do not not have duplicates, we can use a `Dictionary` data structure. The keys in the Dictionary are the IDs of the nodes and its corresponding values are a `List` of `Node` objects.

The methods are

i. `get(key: INTEGER) : List` // returns a `List`

ii. `set(key: INTEGER, data: List) : None` // setting a key value pair in the `Dictionary`.

(d) The idea for the algorithm is as follows:

- We have a `List`, `visited` to keep the nodes found. Set to empty initially.
- Call recursive function to lookup all the directly connected nodes from `source` and return `visited`
  - Base case, if `source` is in `visited` then return `visited`
  - Recursive step:
    - Add `source` to `visited`
    - Lookup `source` from graph for a `List` of neighbour nodes. Call recusive function for each neighbour node as source

We will assume the following method to the List data structure introduced above:

- `inList(n: Object) : Boolean` // returns `True` if object `n` is in the `List`

```
FUNCTION Helper(graph: DICTIONARY, source: INTEGER, visited:
List) RETURNS List
DECLARE neighbours: List

    IF visited.inList(source) THEN
        RETURN visited
    ENDIF

    visited.append(source) //1m
    neighbours <- graph.get(source) //1m
    node <- neighbours.getHead()
    WHILE node <> None DO
        visited <- Helper(graph, node.getID(), visited) //1m
        node <- neighbours.getNext(node)
    ENDWHILE

    RETURN visited
ENDFUNCTION

FUNCTION FindPath(graph: OBJECT, source: INTEGER) RETURNS List
DECLARE visited: List
    visited <- List()
    Helper <- (graph, source, visited)
    RETURN visited
ENDFUNCTION
```

(e) The idea for the algorithm is as follows:

- Iterate over the set of nodes in graph
- Check if nodes visited is same size as the nodes in the graph

We will assume the following method available on List and `DICTIONARY` object:

- `size(): INTEGER` // size of List
- `getKeys(): ARRAY [1:N] of INTEGERS` // array of the keys in `DICTIONARY`

```
PROCEDURE Helper(graph: DICTIONARY)
    nodes <- graph.getKeys()
    FOR i <- 1 to LEN(nodes) //LEN returns size of array
        paths <- FindPath(graph, nodes[i])
        IF paths.size() = LEN(nodes) THEN //1m
            OUTPUT( "Node", nodes[i] )
        ENDIF
    ENDFOR
```

3. (a) i. Given an input data , a hash function should uniformly distribute the input data across the range of addresses.
   ii. The address is fully determined by the data being hashed or given two identical input data the hash function must generate the same address.
   iii. Produce minimal collision

   (b) When the hash function generates a address in the hash table that is already occupied. In separate chaining:

```
IF the address maps to a memory location that is not occupied
THEN
    Create a empty linked list in the memory location
    Append the data value to the linked list
ELSE
    Append the data to the end of the linked list
ENDIF
```

   (c) In linear probing, the data values that are map to the same address in the hash table will have the data stored in consecutive locations, input data that maps to these locations that are already occuied further increase the length of the consecutive locations, resulting in clustering of data in the hash table.

   (d) We have

```
IF table[i] = NONE THEN
    table[i] <- name //(i)
    RETURN True
    ELSE IF i = key THEN
        RETURN FALSE //(ii)
    ELSE
    step <- step + 1 //(iii)
ENDIF
```

   iv. Instead of using the next free memory location, this algorithm selects a new location that is twice the interval from the previous location.
   This solves the clustering problem , as the data are more distributed in the hash table.
   v. The probing does not search for all memory locations in the table. Results in a lot of empty slots that are not used.

4. (a) A set of rules that determine how the sender and receiver exchange data

   (b) Resolve domain names to IP addresses

   (c)
   - The logical name space (data) is centralised, there is only 1 global domain name space on the Internet. ie domain names are unique.
   - The database storing the name space is decentralised via a hierarchical structure. Different partitions of the namespace are maintained by different authoritative DNS servers.

(d)
- `http`: protocol
- `nationaljc.moe.edu.sg`: domain name
- `8088`: port number of listening socket
- `/20SH07`: resource

(e)
- Domain name resolved to ip address
- Browser connects to the remote server port `8088`
- Sends a http request for `/20SH07`

(f)
- Web browser client uses the HTTP protocol to send and receive html contents, which is converted from the raw email content by the email web application server.(Gmail)
- Offline vs online access
- Native device user interface vs web interface
- Native email client uses the SMTP protocol to communicate with a email server

(g)
- Create a socket to connect to the web server ip address and port number (8088)
- Generate a HTTP GET request message for the hidden page resource (/20SH07)
- Receive and decode the return HTML message content
- Parse the HTML content to extract the parts that contains data

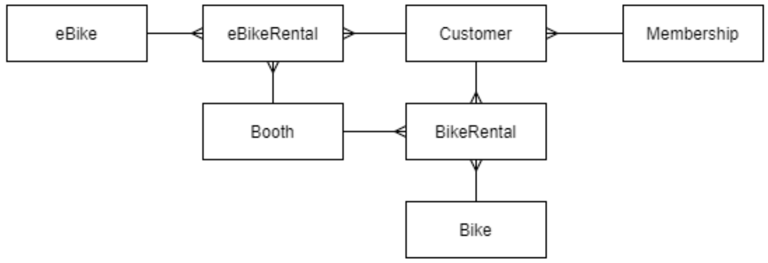
(h)
- The contents of the school's web site is protected by copyrights laws. (you aren't copyrighting a "website," but you can copyright the contents of that website)
- By extracting data from the web site, you are re-using "original content" without permission and therefore is in violation of copyrights laws.
- The privacy of the people whose personal data you extract has been compromised.

(i)
- Consent is not obtained from the staff / students
- Purpose for the use of the data is different from the original intent.
- Data Protection is not inplace
- Removal of contents once its purpose is no longer in need.

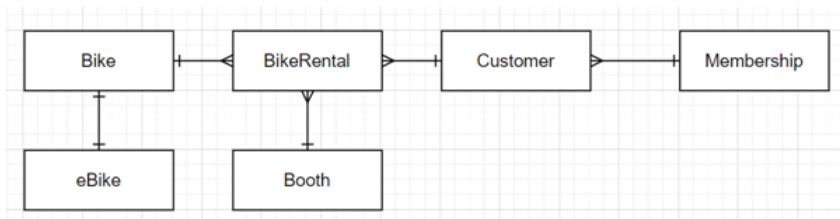
(j) A central authority should kept the information, access to information is granted and control by the central authority. Authentication and access control should be enforced.

(k)
- Encoding is in utf-16
- Web scraping program is using another encoding scheme by default
- Terminal console used by the web scraping program does not have the fonts for the character encoding.

5. (a) The E–R diagram looks like



OR



(b) `Customer (CustID, Name,Contact,CreditCardNumber, Tier*)`

`Bike (SerialNumber, ModelNumber, YOM, ServiceDate,Mileage, eBike: BOOLEAN)`

`eBike (SerialNumber*, Power,BatteryCapacity,Charge)`

`Booth (BoothID, Address)`

`Membership (Tier, MonthlySubscription, RentalRate)`

(c)
- Range Eg date and time
- Format, Eg Credit Card Number
- Presence, Eg already a Customer, BoothID
- Length, Eg Serial Number

(d) The decision table looks like

		Rules							
		1	2	3	4	5	6	7	8
Conditions	Last Serviced Date $>$ 12 months	Y	Y	Y	Y	N	N	N	N
	Mileage $>$ 1000km	Y	Y	N	N	Y	Y	N	N
	e-bike battery $<$ 50% charge	Y	N	Y	N	Y	N	Y	N
Actions	Cannot checkout bike	✓	✓	✓	✓	✓		✓	
	Can check out with Warning message						✓		
	Can check out bike								✓

(e) And the simplified one looks like

Conditions	Last Serviced Date > 12 months	Y	N	N	N
	Mileage > 1000km	-	-	Y	N
	e-bike battery < 50% charge	-	Y	N	N
Actions	Cannot checkout bike	✓	✓		
	Can check out with Warning message			✓	
	Can check out bike				✓

(f) The flowchart looks like