Advanced Higher

Computing Science

Software Design and Development

Implementation Notes

Name:_______________________________

IMPLEMENTATION:
DATA TYPES & STRUCTURES

DATA STRUCTURES: PARALLEL 1 DIMENSIONAL ARRAY

A 1D array is an ordered sequence of simple data types, all of the same type.

Parallel 1D arrays allows related data to be stored together

	Names		Ages		Houses
(0)	Rose	*(0)*	12	*(0)*	Arran
(1)	Jack	*(1)*	14	*(1)*	Bute
(2)	Laura	*(2)*	11	*(2)*	Bute
(3)	James	*(3)*	15	*(3)*	Cumbrae

In the example above, all the information about Rose is stored in index position 0 **in each array**.

It is important, to keep the data together, that related information is entered into the same index position for each array.

DATA STRUCTURES: RECORDS / ARRAY OF RECORDS

Records are **customised data types** created by the programmer. They can contain **several variables** which can be of **different data types**.

When you create a record structure, you are essentially creating a database structure.

Pupils

Field Name	Data Type
Firstname	Text
Surname	Text
Age	Number
House	Text

Record Structure

A **record structure** is created by giving the structure a name and defining the ‘fields’ required.

```
RECORD recordname IS
        {datatype fieldname1, datatype fieldname2, datatype fieldname3…}
```

Array of Records

An **array of records** is then declared which specifies the size of the array and the record structure to use (as the data type):

DECLARE ***arrayname***(indexes) **AS** ***recordname***

Notice, instead of declaring the array using a data type such as integer or string, we have used the name of the record structure as the data type.

Note about Data Structures

You learned about Parallel 1D arrays and Arrays of Records at Higher. You will find coding examples of how these are used in your Higher practical notes.

At Advanced Higher level, these structures are used to store data that:

- Integrates with databases
- Is sorted using sorting algorithms (Insertion Sort / Bubble Sort)
- Is searched using a Binary Search algorithm

2D ARRAYS

A two dimensional array is a data structure consisting of rows and columns. Two dimensional arrays work in exactly the same way as 1D arrays except that instead of just a single index for rows, a column index can also be assigned.

A 2D array can be thought of a as a grid or a table.

9	3	8	6	2	4	1
5	3	2	7	4	1	8
5	3	8	1	9	3	2
5	8	1	6	2	9	2
4	3	1	2	1	8	7
6	4	7	2	8	9	2
3	5	2	1	6	3	2

London	Glasgow	Delhi	Paris
Miami	Berlin	Dundee	Madrid
Lisbon	Rome	Naples	Liverpool
Barcelona	Marseille	Canberra	Montreal

2D arrays can be used to store any data type. However, all data in the array must be set to the **same** data type.

This has implications if a 2D array is used to store details of, for example, city names and temperatures.

London	15
Miami	26
Lisbon	24
Barcelona	31
Glasgow	1
Berlin	17

This 2D array would have to be declared using String as the data type.

This means that the temperature figures would not be stored as values and could not be manipulated as values (e.g. find highest/lowest, sorting etc)

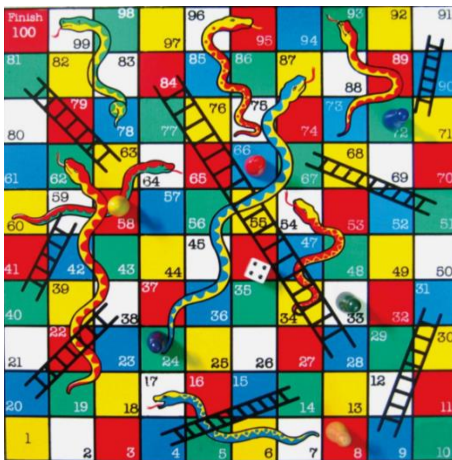
In this case, it may be better to consider the use of two separate 1D arrays.

A record structure would be an even better solution to this problem since it has the following advantages over 1D arrays:

- Data in each column can be referred to using a meaningful identifier, rather than a number.
- Each column can contain data with different data types

The use of a 2D array should always be considered where the data being stored is used to represent a visual grid.

1	6	4						2
2			4		3	9	1	
		5		8		4		7
	9				6	5		
5			1		2			8
		8	9				3	
8		9		4		2		
	7	3	5		9			1
4						6	7	9



Declaring 2D Arrays

When declaring a 2D array, two indices are required. The first specifies the number of rows and second specifies the number of columns.

```
Dim my2Darray(8, 5) As Integer
```

2D arrays declared outside of sub-programs will have a global scope, those inside will be local.

2D Arrays with Standard Algorithms

Standard algorithm code that was previously used on 1D arrays can be easily adapted to work on a 2D array.

```
SET counter TO 0
FOR index FROM 1 TO 10
        IF my1Darray(index) = target THEN
                SET counter TO counter + 1
        END IF
END FOR
```

A counting occurrences algorithm using a 1D array uses a single loop to traverse the rows.

Whereas the same algorithm for a 2D array would require two loops – one to traverse the rows and one to traverse the columns.

```
SET counter TO 0
FOR rows FROM 1 TO 5
        FOR cols FROM 1 TO 10
                IF my2Darray(rows,cols) = target THEN
                        SET counter TO counter + 1
                END IF
        END FOR
END FOR
```

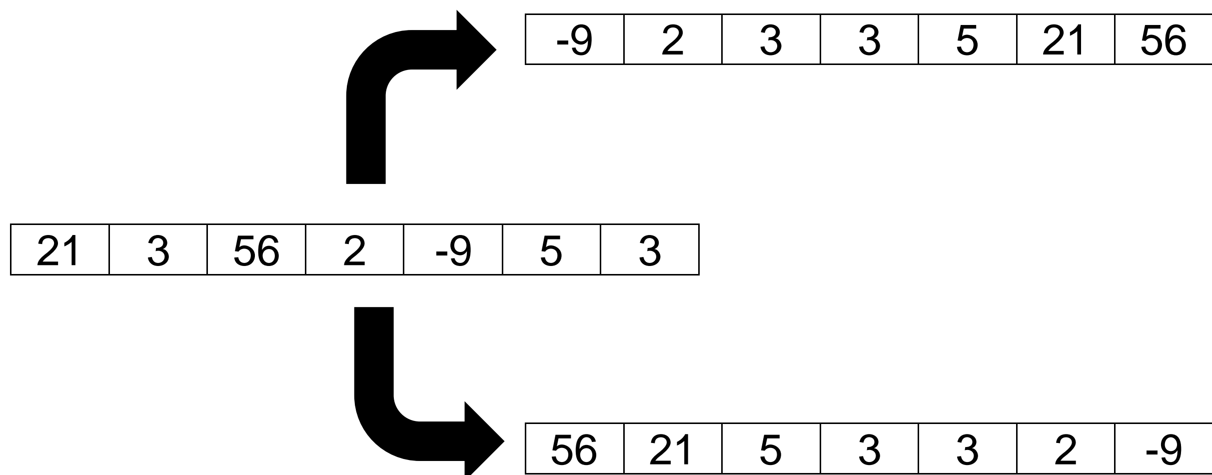
For coding efficiency, it should always be considered whether it is necessary to include all rows and columns in a search for a target value.

It may be that the data being sought can only be present in certain rows or columns and therefore the loops should be adjusted to take this into consideration.

IMPLEMENTATION: STANDARD ALGORITHMS

SORTING ALGORITHMS

Sorting algorithms are used to re-order the items in an array (list) into ascending or descending order.



The two main sorting algorithms you need to understand are:

- **Insertion Sort**
- **Bubble Sort**

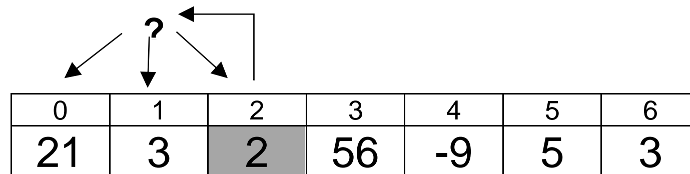
You must be able to write the code for these algorithms in your project and in the exam.

INSERTION SORT

Insertion sort is a comparative sort, which builds the sorted array one entry at a time.

It works by ordering the first two items, then inserting the 3rd item in the correct place, relative to the first two, the 4th item in the correct place, relative to the first 3, and so on.

This means that there is a continual reapplication of comparisons and swaps for each element in the array.



By the end of each traversal, each item will be in the correct position in the list of sorted items so far.

Another way of understanding the steps taken by this algorithm:

Pass 1

- Store value in list position 2 in temporary variable
- Compare temporary value with value in list position 1
- If list position 1 value is larger than temporary value
    - copy list position 1 to list position 2
- If list position 1 value is not larger
    - copy temporary value to list position 2

Pass 2

- Store value in list position 3 in temporary variable
- Compare temporary value with values in list positions 2 and then 1
- Each time list value is larger than temporary value
    - Copy list item to next list position
- If compared list item is not larger than temporary item
    - Copy temporary value to current list position (2 or 1)

Pass 3

- Store value in list position 4 in temporary variable
- Compare temporary value with values in list positions 3 and then 2 and then 1 … and so on.

Insertion Sort – The Algorithm

```
PROCEDURE insertion_sort(list)
     DECLARE value INITIALLY 0
     DECLARE index INITIALLY 0
     FOR i = 1 to length(list)-1 DO
          SET value TO array[i]
          SET index TO i
          WHILE (index > 0) AND (value < list[index-1]) DO
               SET list[index] TO list[index-1]
               SET index TO index - 1
          END WHILE
          SET list[index] TO value
     END FOR
END PROCEDURE
```

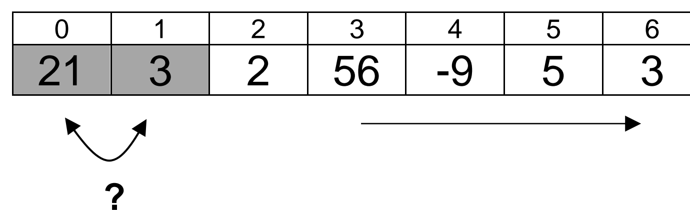
Insertion Sort – Pseudocode and Description

Design	Commentary
`fixed loop i = 1 to length(list)-1`	Loop from the second element to the last
`store the value at array index i`	Store the current temporary value
`store the starting position of the inner loop`	Store the current position in the array — this will be used as a starting point to count backwards during the comparisons
`while index > 0 and value < list[index-1]`	Continue comparing previous values in the list with the temporary value until the start of the array is reached or the two values are in the correct order
`copy the value at index i into index i+1`	The compared value is copied into the element to the right
`reduce the index by 1`	Decrement the element being compared next
`end while`	
`copy the stored value into index i`	The temporarily stored value is copied into the correct place
`end fixed loop`	

BUBBLE SORT

Bubble sort is a comparative sort, which repeatedly steps through the list to be sorted.

During each pass, adjacent items are compared and swapped if they are out of order. This begins by comparing the first item with the second and swapping if first is greater than second. Second and third place is then compared and so on until the end of the pass.



After each traversal, the largest number will have found its way to its correctly sorted position (if sorting in ascending order).

Key features

- Adjacent values are compared, swapping each time a value is out of order
- At the end of each pass, a boolean variable called *swaps* is checked.
- If a swap has taken place during a pass, will have been set to true.
- If no swap takes place during a pass, swaps will remain false and the loop will terminate.

If no swaps take place during a pass, this indicates that the list must have been sorted by the end of the previous pass.

In order to make sure the list is sorted, the algorithm must always complete this final pass with no swaps taking place before the loop can terminate.

Bubble Sort – The Algorithm

```
PROCEDURE bubble_sort(list)
     DECLARE n INITIALLY length(list)
     DECLARE swapped INITIALLY TRUE
     WHILE swapped AND n >= 0
          SET swapped TO False
          FOR i = 0 to n-2 DO
               IF list[i] > list[i+1] THEN
                    SET temp TO list[i]
                    SET list[i] TO list[i+1]
                    SET list[i+1] TO temp
                    SET swapped TO TRUE
               END IF
          END FOR
          SET n TO n - 1
     END WHILE
END PROCEDURE
```

Bubble Sort Algorithm

Bubble Sort – Pseudocode and Description

Design	Commentary
`start conditional loop`	
`set swapped to false`	Set a flag variable to note if a swap has been made
`fixed loop i = 0 to length of list[] - 2`	Loop from the first element to the penultimate element of the array 'list' Note: -2 is equivalent to the second-last element of the array, as the array indexes from 0
`if list[i] > list[i+1] then`	Compare two adjacent values
`swap the two values`	Swap the two values if they are not in the correct order
`set swapped to true`	Set the flag variable to note that a swap has taken place
`end if`	
`end fixed loop`	
`end conditional loop when swapped is equal to false`	The sort ends when a pass results in no swaps

Bubble Sort – Improvement

Consider an array that stores the following values:

0	1	2	3	4	5	6	7	8
45	23	99	7	3	64	37	63	34

After one pass through the array, the largest value will always 'bubble' up to the end of the array.

23	45	7	3	64	37	63	34	99

After a second pass, the second-largest number is also sorted.

23	7	3	45	37	63	34	64	99

When bubble sorting a list of values, the number of iterations carried out by each nested loop can be reduced by one each pass. This improves the efficiency of the bubble sort algorithm.

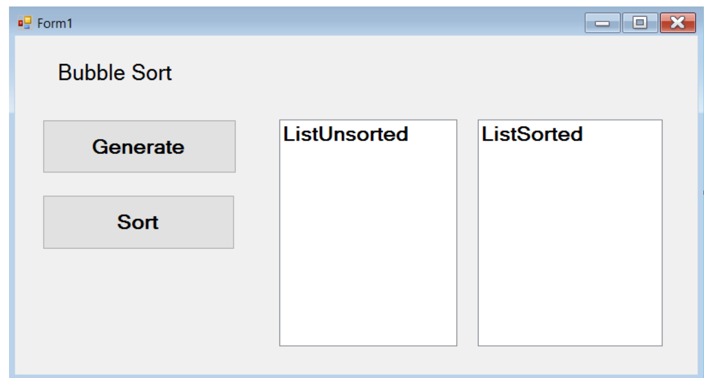
Design	Commentary
`n equals the length of an array called list`	The length of the array is stored in a variable
`start conditional loop`	
`set swapped to false`	
`fixed loop i = 0 to n - 2`	Loop from the first element to the penultimate array element
`if list[i] > list[i+1] then`	
`swap the two values`	
`set swapped to true`	
`end if`	
`end fixed loop`	
`n = n - 1`	Each fixed loop reduces the iterations by 1, as one more element is sorted correctly at the end of the array
`end conditional loop when swapped is equal to false`	

Worked Example 5: **Bubble Sort – Coding (using improvement)**

An implementation of Bubble Sort.

This example demonstrates 10 random numbers being added to a 1D array. The list is then sorted into ascending order using a bubble sort which can detect when the list is sorted.

- Start a new project and create the form shown.




```
Public Class Form1

    Dim Mylist(10) As Integer
```

```
Private Sub Generate_Click(sender As Object, e As EventArgs) Handles Generate.Click

        Randomize()

        For index = 0 To 9
            Mylist(index) = Int(Rnd() * 50) + 1
            ListUnsorted.Items.Add(Mylist(index))
        Next

End Sub
```

```
Private Sub Sort_Click(sender As Object, e As EventArgs) Handles Sort.Click

        Dim inner As Integer
        Dim temp As Integer
        Dim swaps As Boolean
        Dim size As Integer

        size = 10

        Do
            swaps = False
            For inner = 0 To size - 2
                If Mylist(inner) > Mylist(inner + 1) Then
                    temp = Mylist(inner)
                    Mylist(inner) = Mylist(inner + 1)
                    Mylist(inner + 1) = temp
                    swaps = True
                End If
            Next inner
            size = size – 1

        Loop Until swaps = False

        For index = 0 To 9
            ListSorted.Items.Add(Mylist(index))
        Next
End Sub
```

Bubble Sort Algorithm

SEARCH ALGORITHMS

Searching algorithms are used to identify the location of a specified item within an array (list) of items.

BINARY SEARCH

A binary search finds a value by continually halving a **sorted list** until a target is, or is not, found.

The code begins by designating a start (S) point and an end (E) point in the list. These are initially the first and last elements of the array.

From these, the target value positioned in the middle of the sorted list is identified (M=(E-S)/2).

Example: **Target = 8**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
2	5	8	10	11	14	17	25	30	37	38	39	50	51	60	65	77
S								M								E

The algorithm compares the target to the value stored at M and makes one of three decisions:

1. If the middle value is larger than the target, then the target must be in the half of the list that contains smaller values.
2. If the middle value is smaller, the target must be in the larger half of the list.
3. If the middle value is equal to the target, then the position of the target has been identified and the search ends.

If either bullet points 1 or 2 are true, then the start or end are reassigned as required. The middle point is then calculated for the remaining list and the same decision is made again.

Target = 8

2	5	8	10	11	14	17	25	30	37	38	39	50	51	60	65	77
S				M			E									

This is carried out again, until a match is found at M.

Target = 8

2	5	8	10	11	14	17	25	30	37	38	39	50	51	60	65	77
S		M	E													

Binary Search – The Algorithm

Procedure:

```
PROCEDURE binary_search(list,target)
     DECLARE low INITIALLY 0
     DECLARE high INITIALLY length(list)-1
     DECLARE mid INITIALLY 0
     DECLARE found INITIALLY FALSE
     WHILE NOT found AND low <= high
          SET mid TO (low+high)/2
          IF target = list[mid] THEN
               SEND "Found at position"&mid TO DISPLAY
               SET found TO TRUE
          ELSE IF target > list[mid] THEN
               SET low TO mid+1
          ELSE
               SET high TO mid-1
          END IF
     END WHILE
     IF found = FALSE THEN
          SEND "Target not found" TO DISPLAY
     END IF
END PROCEDURE
```

Function:

```
FUNCTION binary_search(list,target) RETURNS INTEGER
     DECLARE low INITIALLY 0
     DECLARE high INITIALLY length(list)-1
     DECLARE mid INITIALLY 0
     DECLARE found INITIALLY FALSE

WHILE NOT found AND low <= high
     SET mid TO (low+high)/2
     IF target = list[mid] THEN
          SET position TO mid
          SET found TO TRUE
     ELSE IF target > list[mid] THEN
          SET low TO mid+1
     ELSE
          SET high TO mid-1
     END IF
END WHILE
RETURN position END FUNCTION
```

Binary Search – Pseudocode and Description

Design	Commentary
low = 0	The lowest index point (S) is stored
high = length(list)-1	The highest index point (E) is stored
found = false	Set a flag variable to show that a match has not yet been found
while not found and low <= high	Conditional loop until the target is found or there are no elements left to examine
set mid = (low + high) / 2	Find the midpoint (M) as halfway between the lowest and highest index
if target = list[mid] then	If a match with the target is found…
set position = mid	…store the position of the match…
set found to true	...and end the conditional loop using the flag variable

else if target > list[mid]
set low = mid + 1
else
set high = mid – 1
end if
	Reset the lowest or highest index depending on whether the target is greater or smaller than the value in the middle index

end while	

If found = false then
Display "not found"
end if
	An optional 'not found' may be added to the end of the algorithm, if required

Linear Search Comparison(Higher Algorithm)

Consider how Linear Search would have performed differently in each of the examples above.

Comparisons made by Linear Search are shown by arrows.

0	1	2	3	4	5	6	7	8	9	10
1	15	21	26	34	39	40	47	55	63	80

So, Linear Search requires **four** comparisons to find 4 and **eight** comparisons to find 7.

This means that on a large, sorted list, Binary Search will perform significantly better than Linear Search.

The only case where Linear Search will give better performance is when the search value is close to the beginning of the list.

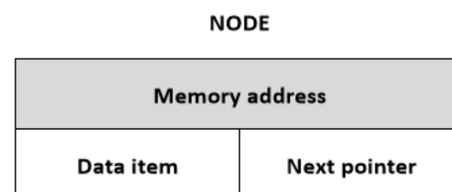
LINKED LISTS (THEORY ONLY)

A linked list is a dynamic data structure. Unlike a 1-D array (which stores each piece of data sequentially in memory), a linked list stores each data item and a pointer (address) to the next data item.

A linked list is a dynamic data structure, as it has no fixed size — it grows and shrinks as required; whereas a 1-D array typically has a set size based on its declaration.

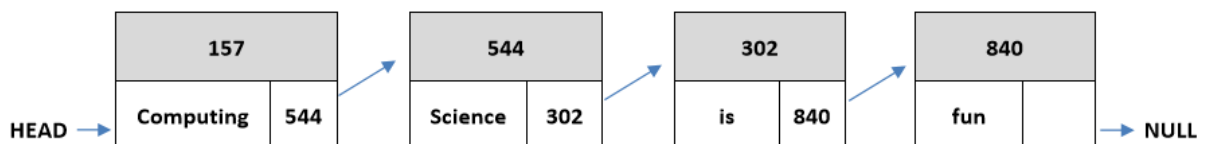
Each element of a linked list is called a NODE. The start of a linked list is called the HEAD and the last element points to the NULL. Each node has its own address in memory, and stores the data item and a pointer to the next node.

The following diagram represents a node.



Singly Linked List

A simple example of a single linked list with four nodes is shown below. The four node linked list stores the words 'Computing', 'Science', 'is', and 'fun'.



Points to note:

- A linked list can store data of multiple data types; a 1-D array is usually limited to one.
- A linked list is a linear data structure; to get to a specific data item, it must always start at the HEAD and work through each node until the data is found.
- A single linked list can only be traversed in one direction — from HEAD to NULL.
- Inserting data into a linked list is more efficient than a 1-D array, as only a pointer is changed rather than shifting the contents of the list (array) into different memory locations.
- Deleting data from a linked list is more efficient than a 1-D array, as only a pointer is changed rather than shifting the contents of the list (array) into different memory locations.

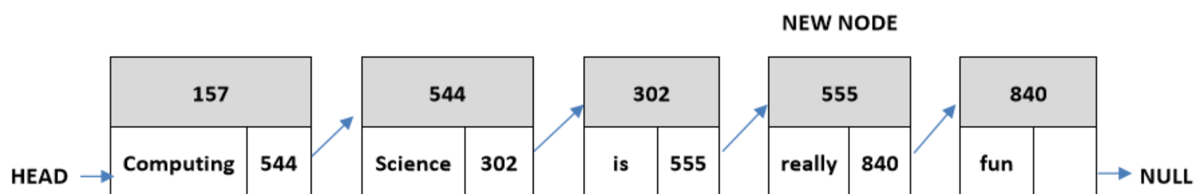
Inserting new data

To insert new data into the list, for example inserting the word ‘really’ between ‘is’ and ‘fun’, a new node is created somewhere in memory and the pointers updated accordingly. To then update the pointer, the list is traversed until ‘is’ is found.

Original

The following is an updated diagram with the word ‘really’ inserted at memory location 555.

Updated

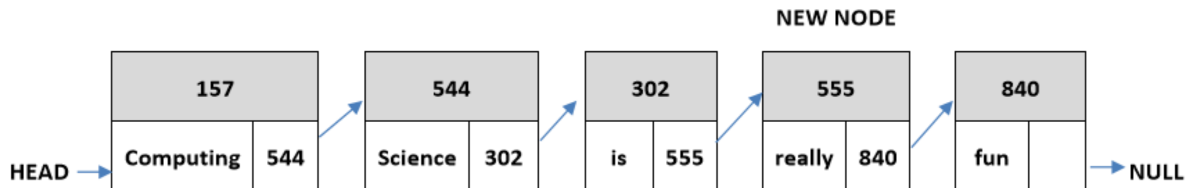


Using a 1-D array data structure and inserting data at a given index means that all data beyond that point is shifted along one location in memory. A linked list is more efficient than a 1-D array, as no data is moved and just one pointer is updated.

Removing data

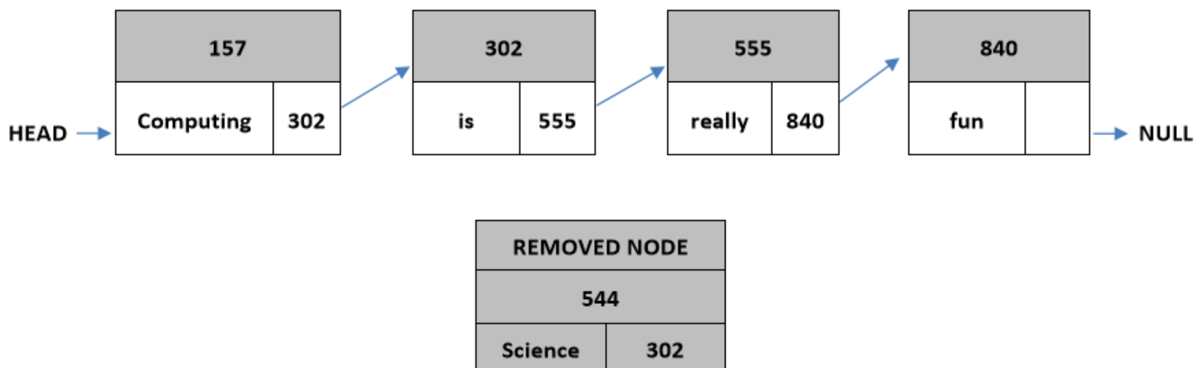
To remove data from the list, for example removing the word ‘Science’, the memory location where the node is stored is freed up and the pointer on the node removed before it is updated. To do this, the list is traversed until the node before ‘Science’ is found.

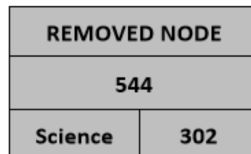
Original



The following is an updated diagram with the word ‘Science’ removed

Updated





Using a 1-D array data structure and removing data at a given index means that all data beyond that point is shifted along one location in memory. A linked list is more efficient than a 1-D array, as no data is moved and just one pointer is updated.

Doubly Linked List

A double linked list is very similar to a single linked list, but has an additional pointer in each node that stores the address of the previous node.

The following diagram represents a node.

Using the same example as for the single linked list, a sample of a double linked list with four nodes is shown below. The four node linked list stores the words 'Computing', 'Science', 'is', and 'fun'.

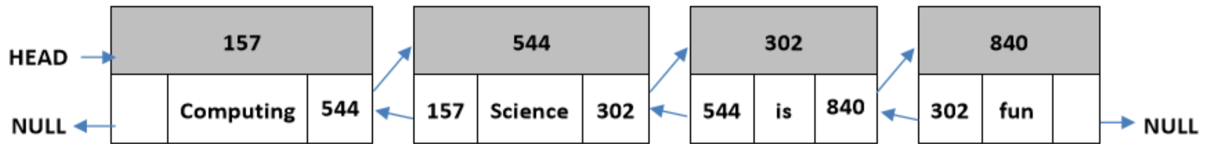
Points to note:

- A double linked list can be traversed in both directions.
- A double linked list requires additional memory, as an extra pointer is being stored on each node.
- If the pointer to the node to be removed is known, then removing a node in a double-linked list is more efficient than in a single linked list: — In a single linked list, to remove a node, the pointer from the previous node is required — to find the pointer, the list is traversed. — In a double linked list, the previous node is determined using the previous pointer.
- To insert a node into a single linked list, the list is traversed until the position is found.
- To insert a node into a double linked list, the list is not traversed if the node is being inserted:
  — at the start of the list
  — at the end of the list
  — after a given node
  or
  — before a given node

Inserting new data

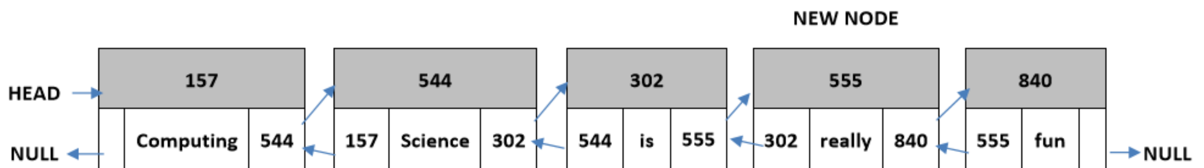
To insert new data into the list, for example the word 'really' to go after the node at address 302, a new node is created somewhere in memory and the pointers before it and after it are updated accordingly.

Original



The following is an updated diagram with the word 'really' inserted at memory location 555.

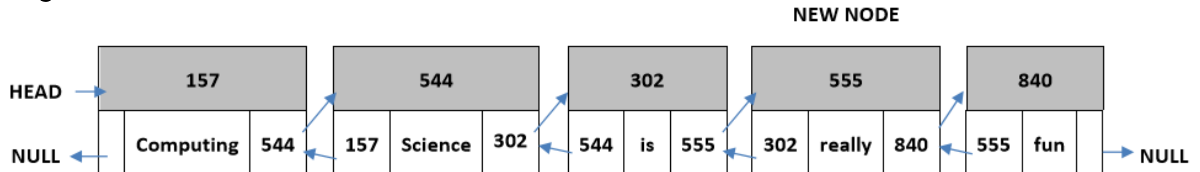
Updated



Removing data

To remove data from the list, for example the word 'Science', the memory location where the node is stored is freed up and the pointer on the node before and after it is updated.

Original



The following is an updated diagram with the word 'Science' removed.

Updated