Introduction to Data Structures

Q.1.Define following terms associated with Data structure and explain classification in detail.

Data Structure

Definition

Data structure is a representation of logical relationship existing between individual elements of data.

Introduction

A data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

Classification of Data Structure

Primitive Data structure

Primitive Data Structures are the basic data structures that directly operated by machine level instructions. They have different memory representations on different computers. All primary data types are known as primitive data structures. Examples: Integers, floating point numbers, character constants, and pointers come under this category.

Non - Primitive Data structure

Non primitive data structures are derived data structure and does not directly operate by machine level instructions. Non-primitive data structures are more complicated data structures than primitive data structures. Non primitive data structures are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Examples: Arrays, lists and files come under this category.

Linear Data structure

A Linear data structure have data elements arranged in sequential manner and each member element is connected to its previous and next element. This connection helps to traverse a linear data structure in a single level and in single run. Such data structures are easy to implement as computer memory is also sequential. Examples of linear data structures are List, Queue, Stack, Array etc.

Non - Linear Data structure

A non-linear data structure has no set sequence of connecting all its elements and each element can have multiple paths to connect to other elements. Such data structures support multi-level storage and often cannot be traversed in single run. Such data structures are not easy to implement but are more efficient in utilizing computer memory. Examples of non-linear data structures are Tree, Graphs etc.

Contiguous Data structure

Contiguous structures can be broken drawn further into two kinds: Those that contain data items of all the same size, and those where the size may differ. The first kind is called the array. In an array, each element is of the same type, and thus has the same size. The second kind of contiguous structure is called structure, in a structure, elements may be of different data types and thus may have different sizes.

Non - Contiguous Data structure

Non-contiguous structures are implemented as a collection of data-items, called nodes, where each node can point to one or more other nodes in the collection. The simplest kind of noncontiguous structure is linked list. A linked list represents a linear, one-dimension type of non-contiguous structure, where there is only the notation of backwards and forwards.

A tree is an example of a two-dimensional non-contiguous structure. Here, there is the notion of up and down and left and right. In a tree each node has only one link that leads into the node and links can only go down the tree. The most general type of non-contiguous structure, called a graph has no such restrictions.

Homogeneous Data structure

Homogeneous data structures are ones that can store a single type of data (numeric, integer, character etc.) Example: Array.

Non - Homogeneous Data structure

Non - Homogeneous data structure are those structures that contains a variety or dissimilar type of data. Example: Structure.

Q.2. Explain advantages of data structure.

Advantages of data structure

1) Data structures allow information storage on hard disks.
2) Provides means for management of large dataset such as databases or internet indexing services.
3) Are necessary for design of efficient algorithms.
4) Allows safe storage of information on a computer. The information is then available for later use and can be used by multiple programs. Additionally, the information is securing and cannot be lost (especially if it is stored on magnetic tapes).
5) Allows the data use and processing on a software system.
6) Allows easier processing of data.
7) Using internet, we can access the data anytime from any connected machine (computer, laptop, tablet, phone etc.)

Q.3. Explain applications of data structure.

Applications of data structure

The data structures store the data according to the mathematical or logical model it is based on. The type of operations on a certain data structure makes it useful for specific tasks. Here is a brief discussion of different applications of various data structures.

Arrays

1) Storing list of data elements belonging to same data type.
2) Auxiliary storage for other data structures.
3) Storage of binary tree elements of fixed count.
4) Storage of matrices.

Linked List

1) Implementing stacks, queues, binary trees and graphs of predefined size.
2) Implement dynamic memory management functions of operating system.
3) Polynomial implementation for mathematical operations
4) Circular linked list is used to implement OS or application functions that require round robin execution of tasks.
5) Doubly linked list is used in the implementation of forward and backward buttons in a browser to move backwards and forward in the opened pages of a website.
6) Circular queue is used to maintain the playing sequence of multiple players in a game.

Stack

1) Temporary storage structure for recursive operations.
2) Evaluation of arithmetic expressions in various programming languages.
3) Conversion of infix expressions into postfix expressions.
4) Checking syntax of expressions in a programming environment.

String

1) In all the problems solutions based on backtracking.
2) Used in depth first search in graph and tree traversal.
3) UNDO and REDO functions in an editor.

Queues

1) It is used in breadth search operation in graphs.
2) Job scheduler operations of OS like a print buffer queue, keyboard buffer queue to store the keys pressed by users.
3) Job scheduling, CPU scheduling, Disk Scheduling.
4) Priority queues are used in file downloading operations in a browser.

Trees

1) Implementing the hierarchical structures in computer systems like directory and file system.
2) Implementing the navigation structure of a website.
3) Decision making in gaming applications.
4) Implementation of priority queues for priority-based OS scheduling functions.
5) Parsing of expressions and statements in programming language compilers.

Q.4. Define algorithm. Write advantages of using algorithms.

Algorithm

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. In other words, an algorithm is sequence of finite steps to complete a task. An algorithm is a procedure for solving a problem. Algorithms are generally created independent of underlying languages, i.e., an algorithm can be implemented in more than one programming language.

Properties of an Algorithm

Key Features

Input - An algorithm should have 0 or more finite number of well-defined inputs.

Output - An algorithm should have 1 or more well-defined outputs, and should match the desired output.

Unambiguous - Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs and outputs should be clear and must lead to only one meaning.

Finiteness - Algorithms must terminate after a finite number of steps.

Feasibility - An algorithm must be feasible. It means it should be possible to implement.

Independent - An algorithm should have step-by-step directions, which should be independent of any programming code.

Correctness - For every input an algorithm should produce correct output.

Example of an algorithm

Write an algorithm to find average of two number.

Algorithm for AVERAGE of two number

Step 1: Take two inputs A and B

Step 2: Calculate sum = A+B

Step 3: Calculate Average = sum/2

Step 4: Print “Average”

Step 5: Finish

Q.5. Define terms: (A) Time complexity (B) Space complexity.

Time Complexity

Time Complexity is a concept in computer science that deals with the quantification of the amount of time taken by a set of code or algorithm to process or run as a function of the amount of input.

Best Case Time Complexity

1) The minimum amount of the time taken by an algorithm on input size n is called best case time complexity.
2) It is denoted by the symbol Big Omega (Ω).
3) It executes in minimum number of steps.
4) Example: The best case time complexity of Quick sort is Ω (nlogn).

Average Case Time Complexity

1) The average amount of the time taken by an algorithm on input size n is called best case time complexity.
2) It is denoted by the symbol Big Theta (Θ).
3) It executes in average number of steps.
4) Example: The average case time complexity of Quick sort is Θ (nlogn).

Worst Case Time Complexity

1) The maximum amount of the time taken by an algorithm on input size n is called best case time complexity.
2) It is denoted by the symbol Big Oh (O).
3) It executes in maximum number of steps.
4) Example: The worst case time complexity of Quick sort is O ($n2$).

Space Complexity

It is the total memory space required by the program for its execution.

Q.6. Explain characteristics of Array.

1) An array holds elements that have the same data type.
2) Array elements are stored in subsequent memory locations.
3) Two-dimensional array elements are stored row by row in subsequent memory locations.
4) Array name represents the address of the starting element.
5) Array size should be mentioned in the declaration. Array size must be a constant expression and not a variable.

Q.7. Explain One-dimensional Array.

Array

One Dimensional Array

An array is defined as an ordered set of similar data items. All the data items of an array are stored in consecutive memory locations in RAM. The elements of an array are of same data type and each item can be accessed using the same name.

Declaration of an array: We know that all the variables are declared before they are used in the program. During declaration, the size of the array has to be specified. The size used during declaration of the array informs the compiler to allocate and reserve the specified memory locations.

Syntax : data_type array_name[n];

Where, n is the number of data items (or) index (or) dimension. 0 to (n-1) is the range of array.

Example : int a[5];
float x[10];

Initialization of Arrays: We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The different types of initializing arrays:

1. At Compile time

(i) Initializing all specified memory locations.
(ii) Partial array initialization
(iii) Initialization without size.
(iv) String initialization.

2. At Run Time

Compile Time Initialization :

We can initialize the elements of arrays in the same way as the ordinary variables when they are declared. The general form of initialization of arrays is

Data_type array_name[size]={ list of values};

(I) Initializing all specified memory locations:

Arrays can be initialized at the time of declaration when their initial values are known in advance. Array elements can be initialized with data items of type int, char etc.

Example : int a[5]={10,15,1,3,20};

(ii) Partial array initialization:

Partial array initialization is possible in c language. If the number of values to be initialized is less than the size of the array, then the elements will be initialized to zero automatically.

Example : int a[5]={10,15};

Initialization with all zeros:

Example : int a[5]={0};

(iii) Initialization without size:

Consider the declaration along with the initialization.

Example : char b[]={'C','O','M','P','U','T','E','R','\0'};

(iv) Array initialization with a string: Consider the declaration with string initialization.

Ex:- char b[]="COMPUTER";

Even though the string "COMPUTER" contains 8 characters, because it is a string it always ends with null character. So, the array size is 9 bytes (i.e., string length 1 byte for null character).

Run Time Initialization:

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. scanf can be used to initialize an array.
Example:
int x[3];
scanf("%d %d %d", &x[0],&x[1],&x[2]);

The above statements will initialize array elements with the values entered through the key board. We can also use for loop like

```
for(i=0;i<3;i++)
Scanf("%d",&x[i]);
```

Display elements of array :

We can display elements of array as,

printf("%d %d %d",x[0],x[1],x[2]);

OR We can also use for loop like
for(i=0;i<3;i++)
printf("%d",x[i]);

Q.8. Explain Two-Dimensional Array.

Two-Dimensional Array
An array consisting of two subscripts is known as two-dimensional array. In two dimensional arrays the array is divided into rows and columns.

Declaration
Syntax: data_type array_name[row_size][column_size];
Example: int arr[3][3];

Where first index value shows the number of the rows and second index value shows the number of the columns in the array.
Initializing two-dimensional arrays:

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces.

Ex: int a[2][3]={0,0,0,1,1,1};

initializes the elements of the first row to zero and the second row to one. The initialization is done row by row.
The above statement can also be written as,

```
int  a[2][3] = {{ 0,0,0},{1,1,1}};
```

by surrounding the elements of each row by braces.

We can also initialize a two-dimensional array in the form of a matrix as shown below

```
int  a[2][3]={
{0,0,0},
{1,1,1}
};
```

When the array is completely initialized with all values, explicitly we need not specify the size of the first dimension.

Example : `int  a[][3] = { {0,2,3}, {2,1,2} };`

If the values are missing in an initializer, they are automatically set to zero.

Example : `int  a[2][3] = { {1,1}, {2} };`
Will initialize the first two elements of the first row to one, the first element of the second row to two and all other elements to zero.

Run Time Initialization :

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. scanf can be used to initialize an array.
Example :

```
int  x[2][3];
scanf("%d %d %d %d %d %d",&x[0][0],&x[0][1],&x[0][2]  ,&x[1][0],&x[1][1],&x[1][2]);
```

The above statements will initialize array elements with the values entered through the key board.

OR
We can also use for loop like,

```
for(i=0;i<2;i++)
{
For(j=0;j<3;j++)
{
Scanf("%d",  &x[i][j]);
}
}
```

Display elements of array:

We can display elements of array as

```
printf("%d%d%d%d%d%d",  x[0][0],x[0][1],x[0][2]  ,x[1][0],x[1][1],x[1][2]); OR
```

We can also use for loop like `for(i=0;i<2;i++)`

```
{
For(j=0;j<3;j++)
{
printf("%d",x[i][j]);
}
}
```

Q.9. Define String. Explain how to declare and initialize string.

Strings

Definition

In C language a string is group of characters (or) array of characters, which is terminated by delimiter \0 (null).

Declaring Strings

C does not support string as a data type. It allows us to represent strings as character arrays. In C, a string variable is any valid C variable name and is always declared as an array of characters.

Syntax: char string_name[size];

The size determines the number of characters in the string name.

Example: char city[10];

char name[30];

Initializing strings

There are several methods to initialize values for string variables.

Example: char city[] = "NEWYORK";

char city[8] ={'N','E','W,'Y','O','R','K,'\0'};

The string city size is 8 but it contains 7 characters and one character space is for NULL terminator.

Q.10. Explain read and write function of string.

String read and write functions

puts()

The C library function int puts(const char *str) writes a string to stdout up to null but not including the null character. A newline character is appended to the output.

int puts(const char *str)

gets()

The C library function char *gets(char *str) reads a line from stdin and stores it into the string pointed to by str.

char *gets(char *str)

putchar()

The C library function int putchar(int char) writes a character (an unsigned char) specified by the argument char to stdout.

int putchar(int char)

getchar()

The C library function int getchar(void) gets a character (an unsigned char) from stdin. This is equivalent to getc with stdin as its argument.
int getchar(void).

Q.11. Explain various string operations.

String Operations

Operations performed on String

Followings are some common operations performed on strings

1. Find length of given string.
2. Compare two strings.
3. Copy one string into other string.
4. Concatenate two strings.
5. Reverse a string.

Algorithm for STRING LENGTH operation

STR_LEN (str)
str: Given string
Step 1: [Initialization]
length = 0
Step 2: [Process until end of the string]
repeat while (str[length] ≠ NULL)
length = length + 1
Step 3: [Finished]
return (length)

Algorithm for STRING COPY operation

STR_CPY (str1, str2)
str1: string1
str2: string2
Step 1: [Initialization]
i = 0
Step 2: [Perform copy]
repeat while (str1[i] ≠ NULL)
str2[i] = str1[i]
i = i + 1
Step 3: [Terminate copied string]
str2[i] = NULL
Step 4: [Finished]

Algorithm for STRING COMPARE operation

STR_CMP (str1, str2)
str1: Given string1
str2: Given string2
Step1: [Initialization]
i = 0
Step2: [Find length of two strings]
length1 = strlen(str1)
length2 = strlen(str2)
Step3: [Compare length of two strings]
if (length1 ≠ length2)
then
write ("Strings are not same")
return()
Step4: [Processed comparison]
repeat while (I < length1)
if (str1[i] ≠ str2[i])
then
write ("Strings are not same")
return()
else
I = I + 1
Step5: [return equal strings]
write ("strings are same")
return()

Algorithm for STRING CONCAT operation

STR_CONCAT (str1, str2)
str1: Given string1
str2: Given string2
Step1: [Initialization]
i = 0
j = 0
Step2: [Process until end of string1]
repeat while (str1[i] ≠ NULL)
str3[i] = str1[i]
i = i + 1
Step 3: [Process until end of string2]
repeat while(str2[j] ≠ NULL)
str3[i] = str2[j]
i = i+ 1
j = j + 1
Step 4: [Terminate string3]
str3[i] = NULL
Step 5: [Finished]
return(str3)

Algorithm for STRING REVERSE operation

STR_REVERSE(str)
str: given string
Step1: [Find length of given string]
length = strlen (str)
Step2: [Initialization]
begin = 0
end = length – 1
Step3: [Process string reverse operation]
repeat while (begin < length)
rev [end] = str [begin]
begin = begin + 1
end = end -1
Step4: [Terminate reverse string]
rev[begin] = NULL
Step5: [Finished]
return(rev)

Stack and Queue

Q.1 Explain stack data structure in detail.

Stack

Stack is a linear list in which insertion and deletion operations are performed at only one end of the list. A stack is generally implemented with only two operations: PUSH and POP. The insertion operation is referred to as a PUSH operation. The deletion operation is referred to as a POP operation.

A pointer TOP keeps track of the top element in stack. Initially when the stack is empty, TOP has a value of zero and when the stack contains a single element: TOP has a value of one and so on. Each time a new element is inserted in the stack the TOP pointer is incremented by one and the pointer is decremented by one when element is deleted. Example – pile of tray in cafeteria.

Q.2 Explain Push and Pop operation of stack with algorithm.

PUSH Operation

To insert an element on the TOP of the stack is called PUSH operation.

Algorithm for PUSH Operation

PUSH(S, TOP, X)
This algorithm inserts an element on top of the stack.
S represents stack.
TOP is a pointer which points to the top of the stack.

```
Step 1: [check for stack overflow]
        if(TOP>=N)
        then write("Stack is Overflow")
        return()
Step 2: [Increment TOP]
        TOP ← TOP +1
Step 3: [Insert element]
        S[TOP] ← X
Step 4: [finished]
        return()
```

POP Operation

To remove an element from the TOP of the stack is called POP operation.

Algorithm for POP Operation

POP (S, TOP)

This algorithm removes an element from the top of the stack.

S represents stack.

TOP is a pointer which points to the top of the stack.

Step 1: [Check for stack underflow]

If (TOP = 0)

then write("Stack is underflow")

return(0)

Step 2: [Delete an element]

Y ← S[TOP]

Step 3: [Decrement pointer]

TOP ← TOP - 1

Step 4: [return deleted element]

return(Y)

Q.3 Explain applications of stack.

Applications of Stack

Three main applications of stack are: (1) Recursion (2) Stack machines (3) Polish notation. The recursive function is called function call to itself. Stack machine provides faster execution of polish notation. Better used for stacking local machines. It is used for representation of arithmetic expression. The process of writing the operators of an expression either before those operands or after operands are called the Polish Notation. There are basically three types of polish notation: Infix, Prefix, Postfix

Recursion

Recursion is a problem solving approach in which a problem is solved using repeatedly applying the same solution to smaller instances. Recursive function is a programming technique that allows the programmer to express operations in terms of themselves. The recursive function is called function call to itself.

Stack machines

Stack machine provides faster execution of polish notation. Better used for stacking local machines.

Polish notation

It is used for representation of arithmetic expression. The process of writing the operators of an expression either before those operands or after operands are called the Polish Notation. There are basically three types of polish notation: Infix, Prefix, Postfix

Q.4 Explain recursive functions with example.

Recursive Functions

Recursion is a problem-solving approach in which a problem is solved using repeatedly applying the same solution to smaller instances. Recursive function is a programming technique that allows the programmer to express operations in terms of themselves. The recursive function is called function call to itself.

Factorial of Number

The factorial function can be recursively defined as,

Factorial(n) = 1 if n = 0
= n * factorial(n-1) if n > 0

Fibonacci series

The function of Fibonacci series can be recursively defined as,

Take n as input

if n=0 or n=1 then, fibo(n) = 0

else if n=2 then, fibo(n) = 1

else fibo(n) = fibo(n-1) + fibo(n-2)

Greatest Common Divisor

The function of GCD can be recursively defined as,

Take two inputs a and b

In case of a ≠ b

if a > b then, gcd(a,b) = gcd(a-b, b)

else if a < b then, gcd(a,b) = gcd(a, b-a)

In case of a = b, gcd(a,b) = a or b

Q.5 Explain steps to convert infix expression to postfix expression using stack.

1) Print operands as they arrive.
2) If stack is empty or contains a left parenthesis on TOP, push the incoming operator onto the stack.
3) If incoming symbol is '(', push it onto stack.
4) If incoming symbol is ')', pop the stack and print operators until left parenthesis is found.
5) If incoming symbol has higher precedence then the TOP of the stack.
6) If incoming symbol has lower precedence then the TOP of the stack, pop and print the TOP. Then test the incoming operator against the new TOP of the stack.
7) If incoming operator has equal precedence with the TOP of the stack, use associativity rule.
8) If expression ends then pop and print all operators of stack.

Q.6 Explain conversion from Infix to Postfix Expression for (a + b) * c – (d - e)

Conversion from Infix to Postfix Expression

(a + b) * c – (d – e)

	INPUT (INFIX)	STACK	OUTPUT (POSTFIX)
1.	(	(	-
2.	a	(	a
3.	+	(+	a
4.	b	(+	ab
5.	)	Empty	ab+
6.	*	*	ab+
7.	c	*	ab+c
8.	-	-	ab+c*
9.	(	- (	ab+c*
10.	d	- (	ab+c*d
11.	-	- (-	ab+c*d
12.	e	- (-	ab+c*de
13.	)	-	ab+c*de-
14.	End	Empty	ab+c*de--

Ans: ab + c * de - -

Q.7 Explain Queue data structure in detail.

Queue

A queue is a linear list of elements in which deletion can take place only at one end, called the FRONT, and insertions can take place only at the other end, called the REAR. Queue is also called First-in-First-out (FIFO) lists.

The term “front” and “rear” are used in describing a linear list only when it is implemented as a queue. Since the first element in a queue will be the first element out of the queue. In other words, the order in which elements enters a queue is the order in which they leave.

Real life examples of Queue are a row of students at a registration center, a movie ticket window example, line of cars waiting to proceed at traffic signal etc.

There are main two ways to implement a queue:
1. Circular queue using array
2. Linked Structures (Pointers)

Primary queue operations:
Enqueue: insert an element at the rear of the queue
Dequeue: remove an element from the front of the queue.

- Figure of Queue:

A queue of people

A computer queue

Q.8 Define Circular Queue data structure.

Circular Queue data structure
A Circular queue is a queue in which data are arranged such that the first element in the queue follows the last element in the queue. When we are deleting an element from simple queue, front pointer is incremented by one and the previous place of front pointer becomes useless. If rear pointer reaches to last element, we cannot insert any more elements. So, wasting of memory is there, this problem is solved by Circular queue.

Q.9 Explain operations of Simple Queue data structure.

Operations on Simple Queue

Insertion in Queue

To insert an element at the REAR end is called INSERTION operation.

Algorithm for INSERT operation

INSERT(Q, F, R, N, Y)

This algorithm inserts an element into the queue.
Q represents queue vector containing N elements.
F and R are pointers pointing to the FRONT and REAR end.
Initially F and R are set to 0.

```
Step 1: [check for Queue overflow]
        if R ≥ N
        then write("Queue is Overflow")
        return(0)
Step 2: [Increment REAR pointer]
        R ← R +1
Step 3: [Insert element]
        Q[R] ← X
Step 4: [Set FRONT pointer]
        if F=0
        then F ←1
      return()
```

Deletion in Queue

To remove an element at the FRONT end is called DELETION operation.

Algorithm for DELETE operation

DELETE(Q, F, R, N)

Q represents queue vector containing N elements.

F and R are pointers pointing to the FRONT and REAR end.

```
Step 1: [Check for Queue underflow]
        if F = 0
        then write("Queue is underflow")
        return(0)
Step 2: [Delete element]
        Y ← Q[F]
Step 3: [Check for Queue empty]
        If F = R
        then F ← 0
             R ← 0
        Else
        F ← F + 1
Step 4: [return element]
        return(Y)
```

Q.10 Explain operations of Circular Queue data structure.

Operations on Circular Queue

Insertion in Circular Queue

To insert an element at the REAR end is called INSERTION operation.

Algorithm for DELETE operation

CQ_INSERT(Q, F, R, N, Y)

This operation inserts an element in the circular queue.
Q represents Queue vector containing N elements.
F and R are pointers pointing to the FRONT and REAR end.
Y is the element to be inserted.
Initially F and R are set to 0.

```
Step 1: [Reset rear pointer]
        if R >= N
        then R ←  1
        else R ←  R + 1
Step 2: [Overflow?]
        if F = R
        then write("QUEUE OVERFLOW")
```

```
        return()
Step 3: [Insert element]
        Q[R] ← Y
Step 4: [Is front pointer properly set?]
        if F=0
        then F ← 1
        return()
```

Deletion in Circular Queue

To remove an element at the FRONT end is called DELETION operation.

Algorithm for DELETE operation

CQ_ DELETE (Q, F, R, N, Y)

This operation deletes an element ad returns it from circular queue.

Q represents Queue vector containing N elements.

F and R are pointers pointing to the FRONT and REAR end.

Y is temporary variable.

Initially F and R are set to 0.

```
Step 1: [Check for Queue underflow]
        if F = 0
        then write("Queue is underflow")
        return(0)
Step 2: [Delete element]
        Y ← Q[F]
Step 3: [Check for Queue empty]
        if F = R
        then F ← 0
        R ← 0
        return(Y)
Step 4: [Increment front pointer]
        if F = N
        then F ← 1
        else F ← F + 1
        return(Y)
```

Q.11 Explain applications of Simple Queue data structure.

Applications of Queue

A queue is the natural data structure for a system to serve its incoming requests. Most of the process scheduling algorithm in operating system uses queues. Queues are used to schedule various jobs, tasks and processes for their execution by the CPU. Simulation is an application of Queue which allows experimenting without modifying the original situation.

Q.12 State the limitations of a Simple Queue and explain Circular Queue data structure.

1) The limitation of simple Queue is that even if we have free spaces, we cannot use these memory spaces. So, simple queue results in wastage of memory. This problem can be solved by using Circular Queue.
2) A Circular queue is a queue in which data are arranged such that the first element in the queue follows the last element in the queue.
3) When we are deleting an element from simple queue, front pointer is incremented by one and the previous place of front pointer becomes useless.
4) If rear pointer reaches to last element, we cannot insert any more elements. So wasting of memory is there, this problem solved by Circular queue.
5) As by moving ahead, when front pointer or rear pointer reaches at last in the next move it moves to the first position.

Linked Lists (LL)

* INTRODUCTION

→ A linked list is a linear collection of data elements. These data elements are called nodes. It can be used to implement stack, queues and other data structures like their variations. (types).

→ A LL can be seen as a TRAIN OR A SEQUENCE OF NODE IN WHICH EACH NODE CONTAINS ONE OR MORE DATA FIELDS AND A POINTER TO THE NEXT NODE.

→ In linked list left part of the node which contains data may include a simple data type, an array or a structure. The right part of the node contains a pointer to the next node. The last node will have no next node connected to it. So it will store a special value called NULL. Since in a linked list every node contains a pointer to another node which is of the same type, it is also called self-referential data type (structure).

→

start (first)

→ L.L contains a pointer variable START (first) that stores the address of the first node in the list. If START = NULL then the LL is empt

* LINKED-LIST REPRESENTATION

⇒ Implementation of LL using following code:

```
struct node
{
    int data;
    struct node * next;
};
```

Eg.

[fig. 1]

NOTE:
-1 denotes the End of List used inst of NULL.

→ Here in above e.g. START is

Pointing to first node and in memory address of first node is 1, the first node contains the address of next node and it is so next node is stored at address 8 and data in it is and so on. Here this uses location 1, 4, 7, 8, 10; 2, 3, 5, 6, 9 are already occupied by some another structures. If we traverse this and print each node then it print 'HELLO'.

3

Linked List versus ARRAYS: (Explain

→ An array is a linear collection of ELEMENTS whereas a linked list is a linear collection of NODES.

→ But unlike an array, a linked list does not store its nodes in consecutive memory locations.

→ In array we can have random access but a linked list does not allow random access of data. Nodes in a L.L can be accessed only in a sequential manner.

→ In LL we can add any number of elements in the List This is not possible in case of an array.

→ Linked List provides an efficient way of storing related data and ~~perform~~ perform basic operations such as insertion, deletion and updating but it need more space than array because of next node.

→ insertion and deletion are easy in in LL than array. but searching time consuming in LL than array.

4

* How actually in memory LL is created and Maintained? i.e Memory allocation & de-allocation

→ We have seen how LL is represented in memory in fig 1. If we want to add a node to an already existing Linked List in the memory we first find free space in the memory and than use it to store the information.

→ for eg. Consider the LL given below in fig.2. The LL contains the roll no and marks obtained by students in DSM subject.

	Roll no	Marks	next
1	IT05	25	-1
2			
START→3	IT07	20	5
4			
5	IT02	15	7
6			
7	IT03	13	8
8	IT04	12	1
9			
10			
11			
12			

→ Here at the times of sharing LL location 2, 4, 6, 9, 10, 11, 12 are occupied but now they are free.

→ Suppose one student was not allowed give exam because of less attendence later point of time he had given exam so

→ now we want to update the list by storing new node in it. At this time it is a question that which part of memory is free (available) and which part is occupied, we need not to worry about it. Operating System by it keep track of free and occupied memory For this OS (the computer OS) maintains a list of all the free memory cells, this list of available space is called

* FREE POOL - Availability Stack (A.S.)

the * free Pool. It has a pointer variable AVAIL which stores the address of the first free space. So Now come to our example, free space for our new node is taken from this list, for this AVAIL pointer is used. Make new pointer and initialize it to AVAIL and then make AVAIL to point to next free space. and using new pointer save roll no. and marks in that node. This was for inserting a new table.

→ Now what if we delete a node as the another LL. when we delete a node that memory which is occupied by node should be added back to the free pool. Hence that free memory is available for use.

* Types of LL (Linked-List)

1. Singly Linked list (S.L.L.)
2. Circular singly Linked List (C.S.L.L.)
3. Doubly Linked List (D.L.L.)
4. Doubly Circular Linked List (D.C.L.L.)

(1.) Singly Linked List :-

→ It is simplest type of LL in which every node contains some data and a pointer to the next node of the same data type.

→ By saying that the node contains a pointer to the next node, we mean that the node stores the address of the next node in sequence.

→ A singly LL allows traversal of data only in one way.

eg.

* Algorithm : "OPERATIONS ON S.L.L."

(1.) Traversing a singly LL.

Step 1 [Initialize]
Set Temp = START

Step 2 While Temp → next != NULL
Temp = Temp → Next
[End of Loop]

Step 4 Exit

(2) Print each node of the LL

Step:1 Set Temp = START

Step:2 while Temp → next != NULL
print Temp → Data
Temp = Temp → Next
[End of Loop]

Step:3 Exit

(3) To Count the member of nodes in LL

Step:1 Set count = 0

Step:2 set temp = START

Step:3 while temp → Next != NULL
set count = count + 1
temp = temp → next
[END of Loop]

Step:6 Exit.

(4) Searching in LL (unsorted)

→ Here Position is taken to print the Position of node and VAL is to be searched.

Step:1 set Temp = START

Step:2 while (Temp → Next != NULL)
If (VAL = Temp → Data)
set Position = temp
Go to step 4
Else
Step:3 set Temp = Temp → Next
[End of If] [End of while]

Step:4 Print Position

Step:5 Exit.

(5) Searching in LL (Sorted)

Step:1 set Temp = Start

Step:2 while (Temp → Next != NULL)

If (Temp → data = VAL) then
set Position = Temp
Go to step 4

Else If (Temp → Data < VAL)
Temp = Temp → Next
Else
Go to step 4

Step: 3 Print position

Step: 4 Exit

(6) Inserting a New Node in a LL

(6.a) Insert at a beginning

→ Allocate memory for new node and initialize its data part.

→ Now the Next Part of new node contains address of START.

→ Now start is node that points to the first node of the list.

Date:
Page No.: 11 / / NOTE

* Algorithm :-

Step 1: If AVAIL = NULL then
Write overflow
Go to step 7
[End of If]

empty
(A.S.)
→ No. space
→ No. free nodes
→ Memory is full
→ So, write overflow

Step 2: SET New-Node = AVAIL

Step 3: SET AVAIL = AVAIL → Next

Step 4: SET New-Node → Data = VAL

Step 5: SET New-Node → Next = START

Step 6: SET START = New-Node

Step 7: Exit.

6.b Insert at the end

* Algorithm :-

Step 1: If AVAIL = NULL then
write overflow
Go to step 9
[End of If]

Step: 2 SET New_Node = AVAIL

Step: 3 Set AVAIL = AVAIL → Next

Step: 4 set New_Node → Data = VAL

Step: 5 set New_Node → Next = NULL

Step: 6 set Temp = START

Step: 7 while (Temp → Next != NULL)
Temp = Temp → next
[End of while]

Step: 8 set temp → next = New_Node

Step: 9 Exit

→ Add the new node after the node pointed by temp. This is done by storing the address of the new node in the next

part of temp.

(G.C.) Insert after a given node

* Algorithm:-

Step 1 If Avail = NULL, then
write overflow
Go to step 16
[End of If]

Step 2 Set New_Node = Avail

Step 3 Set Avail = Avail → Next

Step 4 Set New_Node → Data = VAL

Step 5 set temp = start

Step 6 Set Pretemp = temp

Step 7 Set while (Pretemp → data != NUM)
set Pretemp = temp
set temp = temp → nex
[End of while]

Step 8 Pretemp → Next = New_node

Step!9 Set new_node → Next = temp

Step!10 Exit

(For Understanding)

→ Add node after 2nd node

→ Add the new node in between the nodes pointed by Pretemp & temp.

(6.d) Insert before a given node:

* Algorithm:-

Step!1 If Avail = NULL then
write overflow
go to step!10
[End of If]

Step 12 set New_Node = Avail

Step 13 set Avail = Avail → Next

Step 14 set New_Node → data = VAL

Step 15 set Temp = Start

Step 16 set Pretemp = Temp

Step 17 set while (Temp → data != NUM)
set Pretemp = Temp
set Temp = Temp → Next
(end of while)

Step 18 Pretemp → Next = New_Node

Step 19 set New_Node → Next = Temp

Step 10 Exit

(For Understanding)

→ Add a new node before node that contains the data '4' *

(6.e) Insert in a Sorted linked list

Take two Pointer variables Temp and Pretemp so that initially both point to the START node.

→ Move Pretemp and Temp untill you find the data part of a node whose value is greater than VAL (data of new node). Pretemp will always point to the node just before the node pointed by Temp.

* Algorithm:

Step 1: If Avail = NULL then
write overflow
Go to step 9
[End of If]

Step:2 Set New_Node = Avail

Step:3 Set Avail = Avail → Next

Step:4 set New_Node → data = VAL

Step:5 set Temp = START
set Pretemp = Temp

Step:6 while CPTR → data < VAL
set Pretemp = temp
set temp = temp → next
[End of while]

Step:7 set Pretemp → next = New_Node

Step:8 set New_Node → next = Temp

Step:9 Exit

(7) Delete the Node from a LL

(7.a) Delete the first Node

* Algorithm:

Step:1 If START = NULL, then
write UNDERFLOW
Go to step5
[End of If]

Step:2 set Temp = start

Step:3 set START = START → Next

Step:4 free Temp

Step:5 Exit

7.6 Delete a last Node

→ The pointer variables will be moved to point to the last node of the list, so that the memory occupied by it can be made free. We also store NULL in the last node (which was second to the last node initially)

* Algorithm :-

Step 1 : If START = NULL then
write Underflow
Go to step 6
[End of If]

Step 2 : Set Temp = START

Step 3 : while (Temp → Next != NULL)
Set Pretemp = Temp
Set Temp = Temp → Next
[End of while]

Step 4 : Set Pretemp → Next = NULL

Step 5 : free (Temp)

Step 6 : Exit

(7.C) Delete the node after a given node

Delete after 2nd node i.e. data=3

NOTE

→ The Pointer variable will be moved to point to the nodes that Contain VAL (data=3) and the node succeeding it.

* Algorithm :-

Step:1 If START = NULL then
write underflow
Go to Step 7
[End of If]

Step:2 Set Temp = START

Step:3 Set Preetemp = Temp

Step:4 while (Preetemp → data != NUM)
set Preetemp = temp
Set Temp = temp → Next
[End of while]

Step:5 Set Preetemp → Next = Temp → Next

Step:6 free (Temp)

Step:7 Exit.

☆ Disadvantages of SLL

→ Stepping backward is not Possible

→ link part of last node is not utilis (used

(2) Circular Singly Linked List!

→ There is no storing of NULL values Circular LL in last node's Link part we store the address of first n and hence, it is possible to go anywhere in LL after once you reach the last node.

→ It is used to implement special data structures like circular stack in which two stacks are to be maintained, both front and back stacks. So, circular linked-list can be used to implement it.

→ In memory

	Data	Next
1	L	8
2	H	4
3		
4	E	1
5		
6	O	2
7		
8	L	6
9		

NOTE:
→ In last node instead of NULL we store the address of first node.

☆ Advantages of CSLL

→ stepping backward is possible

→ link part of last node is utilised (used)

⟹ OPERATIONS ON C.S.L.L.

(1) Insertion

(1.a) Inserting a node at beginning

→ Take a pointer 'Temp' that points to the START node of the list.

→ Move Temp to point to last node of List.

→ Add new node between 'Temp' and 'START'.

→ Make the new node to be called as the START node.

* Algorithm:

Step: 1 If AVAIL = NULL then
write over flow
Go to step 10

Step: 2 Set New-Node = Avail

Step: 3 Set Avail = Avail → Next

Step: 4 Set New-Node → Data = VAL

Step 5 Set Temp = START

Step 6 while (Temp → Next != START)
Temp = Temp → Next
[End of while]

Step 7 Set New_Node → Next = START

Step 8 Set Temp → Next = New_Node

Step 9 Set START = New_Node

Step 10 Exit.

(1.b) Inserting at Last

→ Move temp up to last node

→ Add new node after the node pointed by Temp.

* Algorithm :-

Step 1 If Avail = NULL, then
write overflow
Go to step 9
[End of If]

Step 2 Set New-Node = Avail

Step 3 Set Avail = Avail → Next

Step 4 Set New-Node → data = Val

Step 5 Set New-Node → Next = START

Step 6 Set Temp = START

Step 7 While (Temp → Next ≠ START)
Set Temp = Temp → Next
[End of while]

Step 8 Set Temp → Next = New-Node.

Step 9 Exit

2. Deleting a node :-

(2.a) Delete the first node

→ Move Temp upto last node and then make next part of last node Pointing to second node and the first node is freed And make second node as a START Node of List

* Algorithm :-

Step: 1 If START = NULL, then
write under flow
Go to step

Step: 2 set Temp = START

Step: 3 while Temp → Next ! = START
Temp = Temp → Next
[End of while]

Step: 4 set Temp → Next = START → Next

Step: 5 Free (START)

Step: 8 set START = Temp → Next

Step: 7 Exit

(2.b) Delete the last node

→ Move Temp so that it will point to the last ~~temp~~ node of the list. Pretemp will always point to the node preceding Temp

→ Make Pretemp's next Part to store START's address and free the space allocated for temp. Now Pretemp is the last node of list.

* Algorithm:

Step: 1 If START = NULL, then
write UNDERFLOW
Go to stop
[End of IF]

Step:2 set Temp = START

Step:3 while (Temp → next != START)
set Pretemp = Temp
set Temp = Temp → next
[End of while]

Step:4 set Pretemp → Next = START

Step:5 free (Temp)

Step:6 Exit

Doubly Linked List

→ Doubly Linked List also known as two way Linked list is more complex type of linked list which contains a pointer to the next as well as the Previous node in the sequence.

→ Therefore it consists of three parts.

1 data
2 a Pointer to next node (R link)
3 a Pointer to Previous node (L link)

→ In "C" the structure of DLL is,

```
struct node
{
    struct node * prev;
    int data;
    struct node * next;
};
```

Eg.

	L link	Data	R link
1	9	O	6
2			
START→3	(NULL) -1	H	7
4	–	–	–
5			
6	1	O	(NULL) -1
7	3	E	9
8			
9	7	L	1

⟹ OPERATIONS ON D.L.L

☆ Inserting a New node in DLL

(1) Inserting at begining

→ Add an new node before the START node. So New node becomes the first

node of the list.

* Algorithm :-

Step:1 If Avail = NULL then
Pf overflow
Go to step
[End of If]

Step:2 Set New_Node = Avail

Step:3 set Avail = Avail → Next

Step:4 set New_Node → Data = VAL

Step:5 set New_Node → L.link = NULL

Step:6 Set New_Node → R.link = START

Step:7 Set START → L.link = new node

Step:8 set START = New_Node

Step:9 Exit

(2) Insert at the end

→ Make a temp pointer to point to first node and then take it upto last node so now temp is pointing to last node

* Algorithm :-

Step: 1 If Avail = NULL, then Pf overflow
exit ();
[End of If]

Step: 2 Set New-Node = Avail

Step: 3 Set Avail = Avail → Next

Step: 4 Set New-node → Data = VAL

Step: 5 Set New-node → RLink = NULL

Step: 6 Set Temp = START

Step: 7 while (Temp → RLink != NULL)
set Temp = Temp → RLink
[End of while]

Step: 8 set temp → RLink = New_node

Step: 9 set New_node → LLink = Temp

Step: 10 Exit ()

(3) Insert after a given node

→ Take a Temp pointer variable and point to first node of list.

→ Move Temp upto the node after which we want to add new node (suppose here after node '2' whose data is '3' we want to add new node whose data = '5')

* Algorithm:-

NOTE: Comments given in (brackets, not to be considered for writing in exam

Step:1 If Avail = NULL, then
Pf overflow
Exit ();
[End of If]

Step:2 Set New_node = Avail

Step:3 Set Avail = Avail → Next

Step:4 Set New_node → data = VAL ('5')

Step:5 Set Temp = START

Step:6 while (Temp → Data! = NUM (here '3'))
set Temp = Temp → Rlink

Step:7 New_node → Rlink = Temp → Rlink

Step:8 New_node → Llink = Temp

Step:9 Temp → Rlink = New_node

Step:10 New_node → Rlink → Llink = New_node

Step:11 Exit ().

(4.) Insert a new node before given node

→ (suppose we want to add a new node before a node whose data = 8)

→ Make a Temp pointer to point to START node and move it upto the node before which we need to add a new node.

* Algorithm:

Step: 1 If Avail = NULL then
Pf overflow
exit (2)

Step: 2 set New_node = Avail

Step: 4 set = Avail → Next

Step: 5 set New_node → data = VAL (here 8)

Step: 6 set Temp = START

Step: 7 while (Temp → data != NUM) (here NUM = 8)
set Temp = Temp → Rlink
[End of while]

Step: 8 set New-node → Rlink = Temp

Step: 9 set New-node → Llink = Temp → Llink

Step: 10 set Temp → Llink = New_node

Step: 11 set New_node → Llink → Rlink = New_node

Step: 12 Exit ();

(5) Insert in a sorted Doubly Linked List

→ Here in this Linked List we want to add a node with data = 4 which is to be placed between the node whose data = 3 and 5

→ Move Temp until new_node for data is less than data of node,
i.e until 4 < 5

* Algorithm :-

Step:1 If Avail = NULL then
Print overflow
Exit ();

Step:2 set New_node = Avail

Step:3 set Avail = Avail → next

Step:4 set New_node → data = VAL

Step:5 set Temp = START

Step:6 while (Temp → data < VAL)
set Temp = Temp → Rlink

Step:7 set New_node → Llink = Temp

Step:8 set New_node → Llink = Temp → Llink

Step:9 set Temp → Llink = New_node

Step:10. set New_node → Llink → Rlink = New_node

Step:11 Exit ();

☆ Deleting a node from a DLL

(1) Delete the first node

* Algorithm :-

Step 1 If (START = NULL), then
Write : UNDERFLOW
Exit ()
[End of If]

Step 2 set Temp = START

Step 3 set START = START → R.link

Step 4 free (Temp).

Step 5 Exit ();

(2) Delete the last node :-

Take a Temp and make it to point to first and Move it upto last node.

* Algorithm :

Step:1 If START = NULL then
Write underflow
exit()

Step:2 set Temp = START

Step:3 while (Temp → R.link != NULL)
Temp = Temp → Rlink

Step:4 Set Temp → L.link → R.link = NULL

Step:5 free (Temp)

(3) Delete the node after given node

→ Delete a node after 2nd node whose data = 2. Make Temp to point to first node move it upto given node.

* Algorithm:

Step 1: If START = NULL then
Write: underflow
Exit ();

Step 2: set Temp = START;

Step 3: while (Temp → data != VAL)
Temp = Temp → R link

Step 4: set P = Temp → R link

Step 5: set Temp → R link = P → R link

Step 6: set P → R link → L link = Temp

Step 7: free (P);

Step 8: Exit ();

(4) Delete the node before given node

* Algorithm :

Step 1 If START = NULL, then
Write(" underflow
Exit()

Step 2 set Temp = START

Step 3 while (Temp → data != VAL)
Temp = Temp → R.link

Step 4 set P = Temp → L.link

Step 5 set P → L.link → R.link = Temp

Step 6 set Temp → L.link = P → L.link

Step 7 free (P)

Step 8 Exit ();

Introduction:

The I/O subsystem of a computer provides an efficient mode of communication between the central system and the outside environment. It handles all the input-output operations of the computer system.

Peripheral Devices

Input or output devices that are connected to computer are called peripheral devices. These devices are designed to read information into or out of the memory unit upon command from the CPU and are considered to be the part of computer system. These devices are also called peripherals.

For example: Keyboards, display units and printers are common peripheral devices.

There are three types of peripherals:

1.Input peripherals : Allows user input, from the outside world to the computer. Example: Keyboard, Mouse etc.

2. Output peripherals: Allows information output, from the computer to the outside world. Example: Printer, Monitor etc

3. Input-Output peripherals: Allows both input(from outised world to computer) as well as, output(from computer to the outside world). Example: Touch screen etc.

Interfaces

Interface is a shared boundary btween two separate components of the computer system which can be used to attach two or more components to the system for communication purposes.

There are two types of interface:

1. CPU Inteface

2. I/O Interface

Let's understand the I/O Interface in details

Input-Output Interface

Peripherals connected to a computer need special communication links for interfacing with CPU. In computer system, there are special hardware components between the CPU and peripherals to control or manage the input-output transfers. These components are called **input-output interface units** because they provide communication links between processor bus and peripherals. They provide a method for transferring information between internal system and input-output devices.

The I/O interface supports a method by which data is transferred between internal storage and external I/O devices.

I/O Bus and Interface Modules

The I/O bus is the route used for peripheral devices to interact with the computer processor. A typical connection of the I/O bus to I/O devices is shown in the figure.

The I/O bus includes data lines, address lines, and control lines. In any general-purpose computer, the magnetic disk, printer, and keyboard, and display terminal are commonly employed.

Each peripheral unit has an interface unit associated with it. Each interface decodes the control and address received from the I/O bus.

It can describe the address and control received from the peripheral and supports signals for the peripheral controller. It also conducts the transfer of information between peripheral and processor and also integrates the data flow.

The I/O bus is linked to all peripheral interfaces from the processor. The processor locates a device address on the address line to interact with a specific device. Each interface contains an address decoder attached to the I/O bus that monitors the address lines.

When the address is recognized by the interface, it activates the direction between the bus lines and the device that it controls. The interface disables the peripherals whose address does not equivalent to the address in the bus.

An interface receives any of the following four commands −

1. Control − A command control is given to activate the peripheral and to inform its next task. This control command depends on the peripheral, and each peripheral receives its sequence of control commands, depending on its mode of operation.
2. Status − A status command can test multiple test conditions in the interface and the peripheral.
3. Data Output − A data output command creates the interface counter to the command by sending data from the bus to one of its registers.
4. Data Input − The data input command is opposite to the data output command. In data input, the interface gets an element of data from the peripheral and places it in its buffer register.

The internal operations in an individual unit of a digital system are synchronized using clock pulse. It means clock pulse is given to all registers within a unit. And all data transfer among internal registers occurs simultaneously during the occurrence of the clock pulse. Now, suppose any two units of a digital system are designed independently, such as CPU and I/O interface.

If the registers in the I/O interface share a common clock with CPU registers, then transfer between the two units is said to be synchronous. But in most cases, the internal timing in each unit is independent of each other, so each uses its private clock for its internal registers. In this case, the two units are said to be asynchronous to each other, and if data transfer occurs between them, this data transfer is called Asynchronous Data Transfer.

But, the Asynchronous Data Transfer between two independent units requires that control signals be transmitted between the communicating units so that the time can be indicated at which they send data. These two methods can achieve this asynchronous way of data transfer:

Strobe control: A strobe pulse is supplied by one unit to indicate to the other unit when the transfer has to occur.

Handshaking: This method is commonly used to accompany each data item being transferred with a control signal that indicates data in the bus. The unit receiving the data item responds with another signal to acknowledge receipt of the data.

The strobe pulse and handshaking method of asynchronous data transfer is not restricted to I/O transfer. They are used extensively on numerous occasions requiring the transfer of data between two independent units. So, here we consider the transmitting unit as a source and receiving unit as a destination.

For example, the CPU is the source during output or write transfer and the destination unit during input or read transfer.

Therefore, the control sequence during an asynchronous transfer depends on whether the transfer is initiated by the source or by the destination.

So, while discussing each data transfer method asynchronously, you can see the control sequence in both terms when it is initiated by source or by destination. In this way, each data transfer method can be further divided into parts, source initiated and destination initiated.

Asynchronous Data Transfer Methods

The asynchronous data transfer between two independent units requires that control signals be transmitted between the communicating units to indicate when they send the data. Thus, the two methods can achieve the asynchronous way of data transfer.

1. Strobe Control Method

The Strobe Control method of asynchronous data transfer employs a single control line to time each transfer. This control line is also known as a strobe, and it may be achieved either by source or destination, depending on which initiate the transfer.

Source initiated strobe: In the below block diagram, you can see that strobe is initiated by source, and as shown in the timing diagram, the source unit first places the data on the data bus.

(a) Block Diagram

(b) Timing Diagram

After a brief delay to ensure that the data resolve to a stable value, the source activates a strobe pulse. The information on the data bus and strobe control signal remains in the active state for a sufficient time to allow the destination unit to receive the data. The destination unit uses a falling edge of strobe control to transfer the contents of a data bus to one of its internal registers. The source removes the data from the data bus after it disables its strobe pulse. Thus, new valid data will be available only after the strobe is enabled again. In this case, the strobe may be a memory-write control signal from the CPU to a memory unit. The CPU places the word on the data bus and informs the memory unit, which is the destination.

2. Destination initiated strobe: In the below block diagram, you see that the strobe initiated by destination, and in the timing diagram, the destination unit first activates the strobe pulse,informing the source to provide the data.

The source unit responds by placing the requested binary information on the data bus. The data must be valid and remain on the bus long enough for the destination unit to accept it. The falling edge of the strobe pulse can use again to trigger a destination register. The destination unit then disables the strobe. Finally, and source removes the data from the data bus after a determined time interval.

In this case, the strobe may be a memory read control from the CPU to a memory unit. The CPU initiates the read operation to inform the memory, which is a source unit, to place the selected word into the data bus.

Handshaking Method

The strobe method has the disadvantage that the source unit that initiates the transfer has no way of knowing whether the destination has received the data that was placed in the bus. Similarly, a destination unit that initiates the transfer has no way of knowing whether the source unit has placed data on the bus.

So this problem is solved by the handshaking method. The handshaking method introduces a second control signal line that replays the unit that initiates the transfer.

In this method, one control line is in the same direction as the data flow in the bus from the source to the destination. The source unit uses it to inform the destination unit whether there are valid data in the bus.

The other control line is in the other direction from the destination to the source. This is because the destination unit uses it to inform the source whether it can accept data. And in it also, the sequence of control depends on the unit that initiates the transfer. So it means the sequence of control depends on whether the transfer is initiated by source and destination.

Source initiated handshaking: In the below block diagram, you can see that two handshaking lines are "**data valid**", which is generated by the source unit, and "**data accepted**", generated by the destination unit.

The timing diagram shows the timing relationship of the exchange of signals between the two units. The source initiates a transfer by placing data on the bus and enabling its data valid signal. The destination unit then activates the data accepted signal after it accepts the data from the bus. The source unit then disables its valid data signal, which invalidates the data on the bus. After this, the destination unit disables its data accepted signal, and the system goes into its initial state. The source unit does not send the next data item until after the destination unit shows readiness to accept new data by disabling the data accepted signal. This sequence of events described in its sequence diagram, which shows the above sequence in which the system is present at any given time.

Destination initiated handshaking: In the below block diagram, you see that the two handshaking lines are "**data valid**", generated by the source unit, and "**ready for data**" generated by the destination unit.

Note that the name of signal data accepted generated by the destination unit has been changed to ready for data to reflect its new meaning.

The destination transfer is initiated, so the source unit does not place data on the data bus until it receives a ready data signal from the destination unit. After that, the handshaking process is the same as that of the source initiated.

The sequence of events is shown in its sequence diagram, and the timing relationship between signals is shown in its timing diagram. Therefore, the sequence of events in both cases would be identical.

Advantages of Asynchronous Data Transfer

Asynchronous Data Transfer in computer organization has the following advantages, such as:

It is more flexible, and devices can exchange information at their own pace.

In addition, individual data characters can complete themselves so that even if one packet is corrupted, its predecessors and successors will not be affected.

It does not require complex processes by the receiving device. Furthermore, it means that inconsistency in data transfer does not result in a big crisis since the device can keep up with the data stream.

It also makes asynchronous transfers suitable for applications where character data is generated irregularly.

Asynchronous Serial Transmission:

The transfer of data between two units is serial or parallel.

In parallel data transmission, n bit in the message must be transmitted through n separate conductor path. In serial transmission, each bit in the message is sent in sequence one at a time. Parallel transmission is faster but it requires many wires. It is used for short distances and where speed is important. Serial transmission is slower but is less expensive.

In Asynchronous serial transfer, each bit of message is sent a sequence at a time, and binary information is transferred only when it is available. When there is no information to be transferred, line remains idle.

In this technique each character consists of three points :

i. Start bit
ii. Character bit
iii. Stop bit

i. Start Bit- First bit, called start bit is always zero and used to indicate the beginning character.
ii. Stop Bit- Last bit, called stop bit is always one and used to indicate end of characters. Stop bit is always in the 1- state and frame the end of the characters to signify the idle or wait state.
iii. Character Bit- Bits in between the start bit and the stop bit are known as character bits. The character bits always follow the start bit.

Serial Transmission of Asynchronous is done by two ways:

a) Asynchronous Communication Interface

b) First In First out Buffer

Asynchronous Communication Interface:

It works as both a receiver and a transmitter. Its operation is initialized by CPU by sending a byte to the control register. The transmitter register accepts a data byte from CPU through the data bus and transferred to a shift register for serial transmission. The receive portion receives information into another shift register, and when a complete data byte is received it is transferred to receiver register. CPU can select the receiver register to read the byte through the data bus. Data in the status register is used for input and output flags.

First In First Out Buffer (FIFO):

A First In First Out (FIFO) Buffer is a memory unit that stores information in such a manner that the first item is in the item first out. A FIFO buffer comes with separate input and output terminals. The important feature of this buffer is that it can input data and output data at two different rates. When placed between two units, the FIFO can accept data from the source unit at one rate, rate of transfer and deliver the data to the destination unit at another rate. If the source is faster than the destination, the FIFO is useful for source data arrive in bursts that fills out the buffer. FIFO is useful in some applications when data are transferred asynchronously.

Mode of Transfer:

The binary information that is received from an external device is usually stored in the memory unit. The information that is transferred from the CPU to the external device is originated from the memory unit. CPU merely processes the information but the source and target is always the memory unit. Data transfer between CPU and the I/O devices may be done in different modes.

Data transfer to and from the peripherals may be done in any of the three possible ways

2. Programmed I/O.
3. Interrupt- initiated I/O.
4. Direct memory access(DMA).

1. **Programmed I/O:** It is due to the result of the I/O instructions that are written in the computer program. Each data item transfer is initiated by an instruction in the program. Usually the transfer is from a CPU register and memory. In this case it requires constant monitoring by the CPU of the peripheral devices.

Example of Programmed I/O: In this case, the I/O device does not have direct access to the memory unit. A transfer from I/O device to memory requires the execution of several instructions by the CPU, including an input instruction to transfer the data from device to the CPU and store instruction to transfer the data from CPU to memory. In programmed I/O, the CPU stays in the program loop until the I/O unit indicates that it is ready for data transfer. This is a time consuming process since it needlessly keeps the CPU busy. This situation can be avoided by using an interrupt facility. This is discussed below.

2. **Interrupt- initiated I/O:** Since in the above case we saw the CPU is kept busy unnecessarily. This situation can very well be avoided by using an interrupt driven method for data transfer. By using interrupt facility and special commands to inform the interface to issue an interrupt request signal whenever data is available from any device. In the meantime the CPU can proceed for any other program execution. The interface meanwhile keeps monitoring the device. Whenever it is determined that the device is ready for data transfer it initiates an interrupt request signal to the computer. Upon detection of an external interrupt signal the CPU stops momentarily the task that it was already performing, branches to the service program to process the I/O transfer, and then return to the task it was originally performing.

3. **Direct Memory Access**: The data transfer between a fast storage media such as magnetic disk and memory unit is limited by the speed of the CPU. Thus we can allow the peripherals directly communicate with each other using the memory buses, removing the intervention of the CPU. This type of data transfer technique is known as DMA or direct memory access. During DMA the CPU is idle and it has no control over the memory buses. The DMA controller takes over the buses to manage the transfer directly between the I/O devices and the memory unit.

Figure - CPU Bus Signals for DMA Transfer

Bus Request: It is used by the DMA controller to request the CPU to relinquish the control of the buses.

Bus Grant: It is activated by the CPU to inform the external DMA controller that the buses are in high impedance state and the requesting DMA can take control of the buses. Once the DMA has taken the control of the buses it transfers the data. This transfer can take place in many ways.

Types of DMA transfer using DMA controller:

Burst Transfer: DMA returns the bus after complete data transfer. A register is used as a byte count, being decremented for each byte transfer, and upon the byte count reaching zero, the DMAC will release the bus. When the DMAC operates in burst mode, the CPU is halted for the duration of the data transfer.

Steps involved are:
1. Bus grant request time.
2. Transfer the entire block of data at transfer rate of device because the device is usually slow than the speed at which the data can be transferred to CPU.
3. Release the control of the bus back to CPU So, total time taken to transfer the N bytes = Bus grant request time + (N) * (memory transfer rate) + Bus release control time.

Example of Data Transfer:

- When a byte of data is available, the device places it in the I/O bus and enables its data valid line.
- The interface accepts the byte into its data register and enables the data accepted line. The interface sets a bit in the status register to as F or flag bit.
- If the flag is equal to 1, the CPU reads the data from the data register. The flag bit is then cleared to 0 by either the CPU or the interface, depending on how the interface circuits are designed.
- The device can now disable the data valid line, but it will not transfer another byte until the data accepted line is disabled by the interface.

<u>Flow chart:</u>

Fig: Flowchart for CPU program to input data

- The transfer of each byte requires three instructions:
    1. Read the status register.
    2. Check the status of the flag bit and branch to step 1 if not set or to step 3 if set.
    3. Read the data register.

Input-output processor (IOP)

An input-output processor (IOP) is a processor with direct memory access capability. In this, the computer system is divided into a memory unit and number of processors.

Each IOP controls and manage the input-output tasks. The IOP is similar to CPU except that it handles only the details of I/O processing. The IOP can fetch and execute its own instructions. These IOP instructions are designed to manage I/O transfers only.

Block Diagram of I/O Processor

Below is a block diagram of a computer along with various I/O Processors. The memory unit occupies the central position and can communicate with each processor.

The CPU processes the data required for solving the computational tasks. The IOP provides a path for transfer of data between peripherals and memory. The CPU assigns the task of initiating the I/O program.

The IOP operates independent from CPU and transfer data between peripherals and memory.

The communication between the IOP and the devices is similar to the program control method of transfer. And the communication with the memory is similar to the direct memory access method.

In large scale computers, each processor is independent of other processors and any processor can initiate the operation.

The CPU can act as master and the IOP act as slave processor. The CPU assigns the task of initiating operations but it is the IOP, who executes the instructions, and not the CPU. CPU instructions provide operations to start an I/O transfer. The IOP asks for CPU through interrupt.

Instructions that are read from memory by an IOP are also called commands to distinguish them from instructions that are read by CPU. Commands are prepared by programmers and are stored in memory. Command words make the program for IOP. CPU informs the IOP where to find the commands in memory.

CPU – IOP Communication:

There is a communication channel between IOP and CPU to perform task which come under computer architecture. This channel explains the commands executed by IOP and CPU while performing some programs. The CPU do not executes the instructions but it assigns the task of initiating operations, the instructions are executed by IOP. I/O transfer is instructed by CPU. The IOP asks for CPU through interrupt. This channel starts by CPU, by giving "test IOP path" instruction to IOP and then the communication begins as shown in diagram:

between IOP and CPU Whenever CPU gets interrupt from IOP to access memory, it sends test path instruction to IOP. IOP executes and check for status, if the status given to CPU is OK, then CPU gives start instruction to IOP and gives it some control and get back to some another (or same) program, after that IOP is able to access memory for its program. Now IOP start controlling I/O transfer using DMA and create another status report as well. AS soon as this I/O transfer completes IOP once again send interrupt to CPU, CPU again request for status of IOP, IOP check status word from memory location and gives it to CPU. Now CPU check the correctness of status and continues with the same process.

Sorting and Searching techniques

Q.1 Explain Sorting. List out different sorting methods.

Sorting:

Definition

Sorting is the process of arranging elements of a List in particular order either ascending or descending. The Sorting is performed according to the key value of each record. There are number of methods available for performing Sorting. Each one is differed from other. Also, the running time of each method is different, (it is the complexity of an algorithm).

In general, the main task in sorting is basically comparison and interchanging position of elements. There are various types of sorting method available. Sorting method are basically divided into two subcategories.

1. Internal Sort

The Sorting method that does not required external memory for sorting the elements is known as internal sort. It is useful when we must sort fewer amounts of elements.

Example:

1. Bubble Sort
2. Quick Sort
3. Selection Sort
4. Insertion Sort

2. External Sort

The sorting method that required external memory for sorting the element is known as external Sort. It is useful when we must sort large number of elements.

Example:

1. Merge Sort
2. Radix Sort

Q.2 Explain Bubble Sort Method.

Bubble sort is a simple sorting algorithm. Bubble Sort is an example of Internal Sort. Bubble Sort compares all the element one by one and sort them based on their values. If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on.

If we have total n elements, then we need to repeat this process for n-1 times. It is known as bubble sort, because with every complete iteration the largest element in the given array, bubbles up towards the last place or the highest index, just like a water bubble rises up to the water surface.

Sorting takes place by stepping through all the elements one-by-one and comparing it with the adjacent element and swapping them if required.

Example:

Figure: Working of Bubble sort algorithm

Algorithm:

Step 1: Repeat up to Step 3 for I=0 to N-1

Step 2: Repeat step 3 for J=0 to N-I-1

Step 3: If a[J] > a[J+1] then

TEMP → a[J]

a [J] → a[J+1]

a[J+1] → TEMP

Step 4: Exit

Q.3 Explain Selection Sort Method.

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty, and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst-case complexities are of $O(n^2)$, where **n** is the number of items.

Example:

Algorithm:

Step 1: Repeat up to Step 5 for i=0 to N-1

Step 2: Min ← a[i]

Pos ← i

Step 3: Repeat Step 4 for j=i+1 to N

Step 4: if a[j] < Min then

Min ← a[j]

Pos ← j

Step 5: TEMP ← a[i]

a[i] ← a[Pos]

a[Pos] ← TEMP

Step 6: Exit

Q.4 Explain Quick Sort.

Quick Sort method is an efficient Sorting method for large list. It works fine for the list having large number of elements. It uses divide and Conquers Strategy in which a list is divided into two smaller Lists.

First initialize Pivot with index of the list of elements, then Low with index of Pivot+1 of elements and high with index of last element.

Now we scan element from Left to right and compare each element with pivot element. If the scanned element is less, then the pivot element. We scan next element and increment the value of low. Repeat same Procedure until we found the element which is greater than the pivot element.

Now we scan elements from right to left and compare each element with Pivot element if the Scanned element we scanned elements is greater then the pivot elements we scan next element which is less than the Pivot element.

Now we compare the value of low and high. If Low is Less, then High then we interchange the element which are at the index Low and High.

Increment the value of low and decrement the value of high, Repeat above procedure while value of low less than or equal to value of high.

After the Completion of first PASS the entire list of elements is divided into two list first list Contains elements which are less then the pivot elements which are greater than the pivot element. The above Procedure is recursively repeated for the sub lists until all the elements in the lists are Sorted.

The order of Comparison for this method is $O(n\log_n)$.

Partitioning:

1) Pivot=First index
2) I=Pivot +1
3) J=Last index
4) I++ until element > Pivot is found
5) J - - until element <= Pivot is found

If (element is not found less, then Pivot) then

Pivot is on Correct Position then no need to Swap and do Pivot+1

6) Swap a[i] & a[j] and repeat step 4 & 5 until j<=i.
7) If(j<i) Swap pivot & a[j].

Algorithm: Quicksort (a [], low, high)

Step 1: if (low<high)

partitionIndex ← partition(a[],low, high)

Step 2: Quicksort(a[],low,partitionIndex-1)

Step 3: Quicksort(a[],partitionIndex+1,high)

Step 4:Exit

Partition(a,low, high)

Step 1:

pivot ← low

i ← pivot+1

j ← high

step 2: repeat step 3,4 and 5 until j<=i

step 3: i ← i+1 until

a[i]>a[pivot]

step 4: j ← j+1until

a[j] <=a[pivot]

step 5: swap a[i] & a[j]

step 6: if(j<=i)

swap a[pivot] & a[j]

Q.5 Explain Insertion Sort.

Insertion Sort method is a Simple Sorting method, a comparison sort in which the sorted list is built one entry at time.It is much less efficient on large list than more advanced algorithms like quick sort, heap sort or more sort.

However, insertion sort provides several advantages like:

- Simple to implement.
- Efficient for small data set.
- It is adaptive as more efficient for small data set
- It is adaptive, as more efficient when given data are sorted.
- It is stable, as it does not change the the relative order of elements with equal keys.
- It is online, because it can sort a list as it receives list.

Data Structure

A very simple example of insertion sort is when we are planning a game of cards, we are taking cards one by one and arranging them accordingly.

Example:

Algorithm:

Step 1: [initialize i]

I ← 1

Step 2:[traverse through the list and compare elements]

Repeat while (i<n)

KEY ← a[i]

J ← i-1

Repeat while (j <= 0 and KEY < a[j])

a[j+1] ← a[j]

j ← j-1

a[j+1] ← KEY

I ← i+1

Step 3: [finish]

Return ()

Q.6 Explain Merge Sort.

The merge sort algorithm closely follows the divide-and-conquer paradigm. it operates as follows.

Divide: Divide the n-elements sequence to be sorted into two sub-sequences of n/2 elements each recursively.

Conquer: Sort the two sub-sequences recursively using merge sort.

Combine: Merge the two sorted sub-sequences to produce the sorted answer.

Algorithm:

Mergesort(a,i,j)

Step 1:

i ← low

j ← high

Step 2:

Mid ←(i+j)/2

Mergesort(a,i,mid)

Mergesort(a,mid+1,j)

Step 3:

Merge(a, i, mid, j)

Algorithm: Merge(a, i, mid,j)

Step 1:

i ← low

j ← mid+1

k ← low

Step 2:

Repeat Step 3 for i<=mid and j<=high

Step 3:

If(a[i] <= a[j])

b[k] ← a[i]

```
        i ← i++
else
b[k] ← a[j]
j ← j+1
Step 4: k ← k+1
Step 5:
        if(i>mid)
                while(j<=high)
                b[k] ← a[j]
                j ← j+1
k ← k+1
Else
 while(i<=mid)
b[k] ← a[i]
i ← i+1
k ← k+1
step 6: Exit
```

Data Structure

Example:
5,2,4,7,1,3,2,6

Q.7 Explain Radix Sort.

Radix sort is based on the idea that the sorting of the input data is done digit by digit from least significant digit to most significant digit and it uses counting sort as a subroutine to perform sorting.

Counting sort is a linear sorting algorithm with overall time complexity $\Theta(N+K)$ in all cases, where N is the number of elements in the unsorted array and K is the range of input data.

The idea of radix sort is to extend the counting sort algorithm to get a better time complexity when K goes up.

Example: lets consider an unsorted array $A = [101, 1, 20, 50, 9, 98, 27, 153, 35, 899]$ and discuss each step taken to sort the array in ascending order.

Step 1:

Step 2:-

Step 3:-

Sorting on Hundred's place digit

20	50	101	1	153	35	27	98	9	899

0	1	2	3	4	5	6	7	8	9
001	101							899	
009	153								
020									
027									
035									
050									
098									

1	9	20	27	35	50	98	101	153	899

Q.8 Explain following terms with example.

(a) Binary search method

(b) Sequential search method

Binary search method:

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in the sorted form. Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues the sub-array as well until the size of the sub-array reduces to zero.

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

Data Structure

First, we shall determine half of the array by using this formula

Here it is, (0 + 9) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.

Now we compare the value stored at location 4, with the value being searched, i.e., 31. We find that the value at location 4 is 27, which is not a match.

As the value to be searched is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

We change our low to mid + 1 and find the new mid value again. low = mid + 1

mid = (low + high) / 2

Here it is, (5 + 9) / 2 = 7

Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.

The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.

Hence, We change our low to mid -1 and find the new mid value again.

low = mid + 1

mid = (low + high) / 2

Here it is, (5 + 6) / 2 = 5 (integer value of 5.5). So, 5 is the mid of the array.

We find that it is a match.

We conclude that the target value 31 is stored at location 5.

Binary search find the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Sequential search method

Sequential search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that item is returned, otherwise, the search continues till the end of the data collection.

The following is our sorted array and let us assume that we need to search the location of value 33 using sequential search.

Step: 1

Data Structure

Step: 2

Step: 3

Step: 4

Step: 5

Step: 6

Step: 7

Q.9 Write an algorithm for following searching method

Algorithm: BINARY_SEARCH(List, n, x)

List: Array

n: Size of Array

x: Element to be searched.

```
Step 1:[Initialization]
        low ← 0
        high ← n-1
        flag   ←1
Step 2:
        while(low <= high)
        repeat Step 3 to 4
Step 3:
        mid = (low+high)/2
Step 4:
        if (x < List[mid])
        then high  = mid-1
        else if (x > List[mid])
        then low= mid+1
        else if(x  = List[mid])
        then
        write("Search  is successful  at location  : mid+1)
        flag  ← 0
        exit
Step 5:
        if ( flag  = 1)
        then Write("Search  is unsuccessful")
Step 6:[Finished]
        exit
```

Algorithm : SEQUENTIAL_SEARCH(List, n, x)

List : Array

n : Size of Array

x : Element to be searched

Step 1: [Initialization]

i ← 0

flag ←1

Step 2:

repeat step 3 for i = 0, 1, 2, … , n-1

Step 3:

if(x = List[i])

then write("Search is successful at location : i + 1")

flag ← 0

exit

Step 4:

if (flag =1)

then Write("Search is unsuccessful")

Step 5: [Finished]

Exit