DATA STRUCTURES WITH C
(Common to CSE & ISE)

Subject Code: 10CS35 **I.A. Marks : 25**
Hours/Week : 04 **Exam Hours: 03**
Total Hours : 52 **Exam Marks: 100**

PART – A

UNIT - 1 8 Hours
BASIC CONCEPTS: Pointers and Dynamic Memory Allocation, Algorithm Specification, Data Abstraction, Performance Analysis, Performance Measurement

UNIT -2 6 Hours
ARRAYS and STRUCTURES: Arrays, Dynamically Allocated Arrays, Structures and Unions, Polynomials, sparse Matrices, Representation of Multidimensional Arrays

UNIT - 3 6 Hours
STACKS AND QUEUES: Stacks, Stacks Using Dynamic Arrays, Queues, Circular Queues Using Dynamic Arrays, Evaluation of Expressions, Multiple Stacks and Queues.

UNIT - 4 6 Hours
LINKED LISTS: Singly Linked lists and Chains, Representing Chains in C, Linked Stacks and Queues, Polynomials, Additional List operations, Sparse Matrices, Doubly Linked Lists

PART - B

UNIT - 5 6 Hours

TREES – 1: Introduction, Binary Trees, Binary Tree Traversals, Threaded Binary Trees, Heaps.

UNIT – 6 6 Hours
TREES – 2, GRAPHS: Binary Search Trees, Selection Trees, Forests, Representation of Disjoint Sets, Counting Binary Trees, The Graph Abstract Data Type.

UNIT - 7 6 Hours
PRIORITY QUEUES Single- and Double-Ended Priority Queues, Leftist Trees, Binomial Heaps, Fibonacci Heaps, Pairing Heaps.

UNIT - 8 8 Hours
EFFICIENT BINARY SEARCH TREES: Optimal Binary Search Trees, AVL Trees, Red-Black Trees, Splay Trees.

Text Book:
1. Horowitz, Sahni, Anderson-Freed: Fundamentals of Data Structures in C, 2nd Edition, Universities Press, 2007. (Chapters 1, 2.1 to 2.6, 3, 4, 5.1 to 5.3, 5.5 to 5.11, 6.1, 9.1 to 9.5, 10)

Reference Books:
1. Yedidyah, Augenstein, Tannenbaum: Data Structures Using C and C++, 2nd Edition, Pearson Education, 2003.
2. Debasis Samanta: Classic Data Structures, 2nd Edition, PHI, 2009.
3. Richard F. Gilberg and Behrouz A. Forouzan: Data Structures A Pseudocode Approach with C, Cengage Learning, 2005.
4. Robert Kruse & Bruce Leung: Data Structures & Program Design in C, Pearson Education, 2007.

TABLE OF CONTENTS

UNIT – 1: BASIC CONCEPTS

1.1-Pointers and Dynamic Memory Allocation:

In computer science, a pointer is a programming language data type whose value refers directly to (or "points to") another value stored elsewhere in the computer memory using its address. For high-level programming languages, pointers effectively take the place of general purpose registers in low-level languages such as assembly language or machine code, but may be in available memory. A pointer references a location in memory, and obtaining the value at the location a pointer refers to is known as dereferencing the pointer. A pointer is a simple, more concrete implementation of the more abstract reference data type. Several languages support some type of pointer, although some have more restrictions on their use than others.

Pointers to data significantly improve performance for repetitive operations such as traversing strings, lookup tables, control tables and tree structures. In particular, it is often much cheaper in time and space to copy and dereference pointers than it is to copy and access the data to which the pointers point.Pointers are also used to hold the addresses of entry points for called subroutines in procedural programming and for run-time linking to dynamic link libraries (DLLs). In object-oriented programming, pointers to functions are used for binding methods, often using what are called virtual method tables.

Declaring a pointer variable is quite similar to declaring an normal variable all you have to do is to insert a star '*' operator before it.

General form of pointer declaration is -

type* name;

where type represent the type to which pointer thinks it is pointing to.

Pointers to machine defined as well as user-defined types can be made

Pointer Intialization: variable_type *pointer_name = 0;

or

variable_type *pointer_name = NULL;

char *pointer_name = "string value here";

The operator that gets the value from pointer variable is * (indirection operator). This is called the reference to pointer.

P=&a

So the pointer p has address of a and the value that that contained in that address can be accessed by : *p

So the operations done over it can be explained as below:

a++;

a=a+1;

*p=*p+1;

(*p)++:

While "pointer" has been used to refer to references in general, it more properly applies to data structures whose interface explicitly allows the pointer to be manipulated (arithmetically via *pointer arithmetic*) as a memory address, as opposed to a magic cookie or capability where this is not possible.

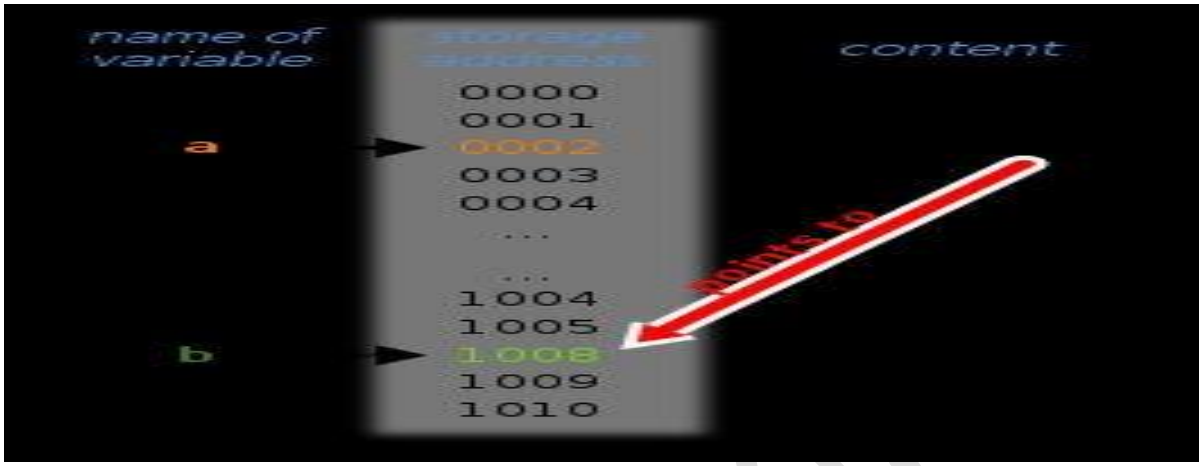


Fig 1: Pointer *a* pointing to the memory address associated with variable *b*. Note that in this particular diagram, the computing architecture uses the same address space and data primitive for both pointers and non-pointers; this need not be the case.

Pointers and Dynamic Memory Allocation:

Although arrays are good things, we cannot adjust the size of them in the middle of the program. If our array is too *small* - our program will fail for large data. If our array is too *big* - we waste a lot of space, again restricting what we can do. The right solution is to build the data structure from small pieces, and add a new piece whenever we need to make it larger. *Pointers* are the connections which hold these pieces together!

Pointers in Real Life

In many ways, telephone numbers serve as pointers in today's society. To contact someone, you do not have to carry them with you at all times. *All you need is their number.* Many different people can all have your number simultaneously. *All you need do is copy the pointer.* More complicated structures can be built by combining pointers. *For example, phone trees or directory information.* Addresses are a more physically correct analogy for pointers, since they really are memory addresses.

Linked Data Structures

All the dynamic data structures we will build have certain shared properties. We need a pointer to the entire object so we can find it. Note that this is a pointer, not a cell. Each cell contains one or more data fields, which is what we want to store. Each cell contains a pointer field to at least one ``next" cell. Thus much of the space used in linked data structures is not data! We must be able to detect the end of the data structure. This is why we need the NIL pointers.

There are four functions defined in c standard for dynamic memmory allocation - calloc, free, malloc and realloc. But in the heart of DMA there are only 2 of them malloc and free. Malloc stands for memmory allocations and is used to allocate memmory from the heap while free is used to return allocated memmory from malloc back to heap. Both these functions uses a standard library header

<stdlib.h> .Warning !!! - free ( ) function should be used to free memmory only allocated previously from malloc, realloc or calloc. Freeing a random or undefined or compiler allocated memmory can lead to severe damage to the O.S., Compiler and Computer Hardware Itself, in form of nasty system crashes.

The prototype of malloc ( ) function is -

void *malloc (size_t number_of_bytes)

Important thing to nore is malloc return a void pointer which can be converted to any pointer type as explained in previous points. Also size_t is a special type of unsigned integer defined in <stdlib.h> capable of storing largest memmory size that can be allocated using DMA, number_of_bytes is a value of type size_t generally a integer indicating the amount of memmory to be allocated. Function malloc ( ) will be returning a null pointer if memmory allocation fails and will return a pointer to first region of memmory allocated when succsefull. It is also recommended you check the pointer returned for failure in allocation before using the returned memmory for increasing stability of your program, generally programmers provide some error handling code in case of failures. Also this returned pointer never needs a typecast in C since it is a void pointer, it is a good practice to do one since it is required by C++ and will produce a error if you used C++ compiler for compilation.Another commonly used operator used with malloc is sizeof operator which is used to calculate the value of number_of_bytes by determing the size of the compiler as well as user defined types and variables.

The prototype of free ( ) function is -

void free (void *p)

Function free ( ) is opposite of malloc and is used to return memmory previously allocated by other DMA functions. Also only memmory allocated using DMA should be free using free () otherwise you may corrupt your memmory allocation system at minimum.

C Source code shown below shows simple method of using dynamic memmory allocation elegantly –

```
#include <stdio.h>

#include <stdlib.h>

int main ()

{

int *p;

p = (int *) malloc ( sizeof (int) ); //Dynamic Memmory Allocation

if (p == NULL) //Incase of memmory allocation failure execute the error handling code block

{

 printf ("\nOut of Memmory");

  exit (1);

}

*p = 100;
```

```
printf ("\n p = %d", *p); //Display 100 ofcourse.

return 0;

}
```

Dynamic Allocation: To get dynamic allocation, use new:

```
p := New(ptype);
```

New(ptype) allocates enough space to store exactly one object of the type ptype. Further, it returns a pointer to this empty cell. Before a new or otherwise explicit initialization, a pointer variable has an arbitrary value which points to *trouble*!

Warning - initialize all pointers before use. Since you cannot initialize them to explicit constants, your only choices are

NIL - meaning explicitly nothing.

New(ptype) - a fresh chunk of memory.

Pointer Examples

Example: P: = new(node); q := new(node);

p.x grants access to the field *x* of the record pointed to by *p*.

```
p^.info := "music";

q^.next := nil;
```

The pointer value itself may be copied, which does not change any of the other fields.

Note this difference between assigning pointers and what they point to.

```
p := q;
```

We get a real mess. We have completely lost access to music and can't get it back! Pointers are *unidirectional*.

Alternatively, we could copy the object being pointed to instead of the pointer itself.

```
p^ := q^;
```

What happens in each case if we now did:

```
p^.info := "data structures";
```

Where Does the Space Come From?

Can we really get as much memory as we want without limit just by using New?

No, because there are the physical limits imposed by the size of the memory of the computer we are using. Usually Modula-3 systems let the dynamic memory come from the ``other side" of the ``activation record stack" used to maintain procedure calls.Just as the stack reuses memory when a procedure exits, dynamic storage must be recycled when we don't need it anymore.

Garbage Collection

The Modula-3 system is constantly keeping watch on the dynamic memory which it has allocated, making sure that *something* is still pointing to it. If not, there is no way for you to get access to it, so the space might as well be recycled. The *garbage collector* automatically frees up the memory which has nothing pointing to it. It frees you from having to worry about explicitly freeing memory, at the cost of leaving certain structures which it can't figure out are really garbage, such as a circular list.

Explicit Deallocation

Although certain languages like Modula-3 and Java support garbage collection, others like C++ require you to explicitly deallocate memory when you don't need it.

Dispose(p) is the opposite of New - it takes the object which is pointed to by *p* and makes it available for reuse. Note that each dispose takes care of only one cell in a list. To dispose of an entire linked structure we must do it one cell as a time. Note we can get into trouble with *dispose*:

Of course, it is too late to dispose of music, so it will endure forever without garbage collection. Suppose we *dispose(p),* and later allocation more dynamic memory with *new*. The cell we disposed of might be reused. Now what does *q* point to?

Answer - the same location, but it means something else! So called *dangling references* are a horrible error, and are the main reason why Modula-3 supports garbage collection. A dangling reference is like a friend left with your old phone number after you move. Reach out and touch someone - eliminate dangling references!

Security in Java

It is possible to explicitly dispose of memory in Modula-3 when it is really necessary, but it is strongly discouraged. Java does not allow one to do such operations on pointers at all. The reason is *security*.Pointers allow you access to raw memory locations. In the hands of skilled but evil people, unchecked access to pointers permits you to modify the operating system's or other people's memory contents.

1.2. Algorithm Specification:

A pragmatic approach to algorithm specification and verification is presented. The language AL provides a level of abstraction between a mathematical specification notation and a programming language, supporting compact but expressive algorithm description.

Proofs of correctness about algorithms written in AL can be done via an embedding of the semantics of the language in a proof system; implementations of algorithms can be done through translation to standard programming languages.

The proofs of correctness are more tractable than direct verification of programming language code; descriptions in AL are more easily related to executable programs than standard mathematical specifications. AL provides an independent, portable description which can be related to different proof systems and different programming languages.

Several interfaces have been explored and tools for fully automatic translation of AL specifications into the HOL logic and Standard ML executable code have been implemented. A substantial case study uses AL as the common specification language from which both the formal proofs of correctness and executable code have been produced.

1.3. Data Abstraction

Abstraction is the process by which data and programs are defined with a representation similar to its meaning (semantics), while hiding away the implementation details. Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time. A system can have several *abstraction layers* whereby different meanings and amounts of detail are exposed to the programmer. For example, low-level abstraction layers expose details of the hardware where the program is run, while high-level layers deal with the business logic of the program.

The following English definition of abstraction helps to understand how this term applies to computer science, IT and objects:

***abstraction - a concept or idea not associated with any specific instance*[1]**

Abstraction captures only those detail about an object that are relevant to the current perspective. The concept originated by analogy with abstraction in mathematics. The mathematical technique of abstraction begins with mathematical definitions, making it a more technical approach than the general concept of abstraction in philosophy. For example, in both computing and in mathematics, numbers are concepts in the programming languages, as founded in mathematics. Implementation details depend on the hardware and software, but this is not a restriction because the computing concept of number is still based on the mathematical concept.

In computer programming, abstraction can apply to control or to data: Control abstraction is the abstraction of actions while data abstraction is that of data structures. Control abstraction involves the use of subprograms and related concepts control flows. Data abstraction allows handling data bits in meaningful ways. For example, it is the basic motivation behind datatype. One can regard the notion of an object (from object-oriented programming) as an attempt to combine abstractions of data and code. The same abstract definition can be used as a common interface for a family of objects with different implementations and behaviors but which share the same meaning. The inheritance mechanism in object- oriented programming can be used to define an abstract class as the common interface.

Data abstraction enforces a clear separation between the *abstract* properties of a data type and the *concrete* details of its implementation. The abstract properties are those that are visible to client code that makes use of the data type—the *interface* to the data type—while the concrete implementation is kept entirely private, and indeed can change, for example to incorporate efficiency improvements over time. The idea is that such changes are not supposed to have any impact on client code, since they involve no difference in the abstract behaviour.

For example, one could define an abstract data type called *lookup table* which uniquely associates *keys* with *values*, and in which values may be retrieved by specifying their corresponding keys. Such a lookup table may be implemented in various ways: as a hash table, a binary search tree, or even a simple linear list of (key:value) pairs. As far as client code is concerned, the abstract properties of the type are the same in each case. Of course, this all relies on getting the details of the interface right in the first place, since any changes there can have major impacts on client code. As one way to look at this: the interface forms a *contract* on agreed behaviour between the data type and client code; anything not spelled out in the contract is subject to change without notice.

Languages that implement data abstraction include Ada and Modula-2. Object-oriented languages are commonly claimed to offer data abstraction; however, their inheritance concept tends to put information in the interface that more properly belongs in the implementation; thus, changes to such information ends up impacting client code, leading directly to the Fragile binary interface problem.

1.4. Performance Analysis:

Performance analysis involves gathering formal and informal data to help customers and sponsors define and achieve their goals. Performance analysis uncovers several perspectives on a problem or opportunity, determining any and all drivers towards or barriers to successful performance, and proposing a solution system based on what is discovered.

A lighter definition is:

Performance analysis is the front end of the front end. It's what we do to figure out what to do. Some synonyms are planning, scoping, auditing, and diagnostics.

What does a performance analyst do?

Here's a list of some of the things you *may* be doing as part of a performance analysis:

- Interviewing a sponsor
- Reading the annual report
- Chatting at lunch with a group of customer service representatives
- Reading the organization's policy on customer service, focusing particularly on the recognition and incentive aspects
- Listening to audiotapes associates with customer service complaints
- Leading a focus group with supervisors
- Interviewing some randomly drawn representatives
- Reviewing the call log
- Reading an article in a professional journal on the subject of customer service performance improvement
- Chatting at the supermarket with somebody who is a customer, who wants to tell you about her experience with customer service

We distinguish three basic steps in the performance analysis process:

- **data collection,**
- **data transformation, and**
- **data visualization.**
  Data collection is the process by which data about program performance are obtained from an executing program. Data are normally collected in a file, either during or after execution, although in some situations it may be presented to the user in real time.

Three basic data collection techniques can be distinguished:

Profiles record the amount of time spent in different parts of a program. This information, though minimal, is often invaluable for highlighting performance problems. Profiles typically are gathered automatically.

Counters record either frequencies of events or cumulative times. The insertion of counters may require some programmer intervention.

Event traces record each occurrence of various specified events, thus typically producing a large amount of data. Traces can be produced either automatically or with programmer intervention.

The raw data produced by profiles, counters, or traces are rarely in the form required to answer performance questions. Hence, *data transformations* are applied, often with the goal of reducing total data volume. Transformations can be used to determine mean values or other higher-order statistics or to extract profile and counter data from traces. For example, a profile recording the time spent in each subroutine on each processor might be transformed to determine the mean time spent in each subroutine on each processor, and the standard deviation from this mean. Similarly, a trace can be processed to produce a histogram giving the distribution of message sizes. Each of the various performance tools described in subsequent sections incorporates some set of built-in transformations; more specialized transformation can also be coded by the programmer.

Parallel performance data are inherently multidimensional, consisting of execution times, communication costs, and so on, for multiple program components, on different processors, and for different problem sizes. Although data reduction techniques can be used in some situations to compress performance data to scalar values, it is often necessary to be able to explore the raw multidimensional data. As is well known in computational science and engineering, this process can benefit enormously from the use of *data visualization* techniques. Both conventional and more specialized display techniques can be applied to performance data.

As we shall see, a wide variety of data collection, transformation, and visualization tools are available. When selecting a tool for a particular task, the following issues should be considered:

Accuracy. In general, performance data obtained using sampling techniques are less accurate than data obtained by using counters or timers. In the case of timers, the accuracy of the clock must be taken into account.

Simplicity. The best tools in many circumstances are those that collect data automatically, with little or no programmer intervention, and that provide convenient analysis capabilities.

Flexibility. A flexible tool can be extended easily to collect additional performance data or to provide different views of the same data. Flexibility and simplicity are often opposing requirements.

Intrusiveness. Unless a computer provides hardware support, performance data collection inevitably introduces some overhead. We need to be aware of this overhead and account for it when analyzing data.

Abstraction. A good performance tool allows data to be examined at a level of abstraction appropriate for the programming model of the parallel program. For example, when analyzing an execution trace from a message-passing program, we probably wish to see individual messages, particularly if they can be related to send and receive statements in the source program. However, this presentation is probably *not* appropriate when studying a data-parallel program, even if compilation generates a message-passing program. Instead, we would like to see communication costs related to data-parallel program statements.

1.5. Performance Measurement:

Performance measurement is the process whereby an organization establishes the parameters within which programs, investments, and acquisitions are reaching the desired results.

Good Performance Measures:

Provide a way to see if our strategy is working

Focus employees' attention on what matters most to success

Allow measurement of accomplishments, not just of the work that is performed

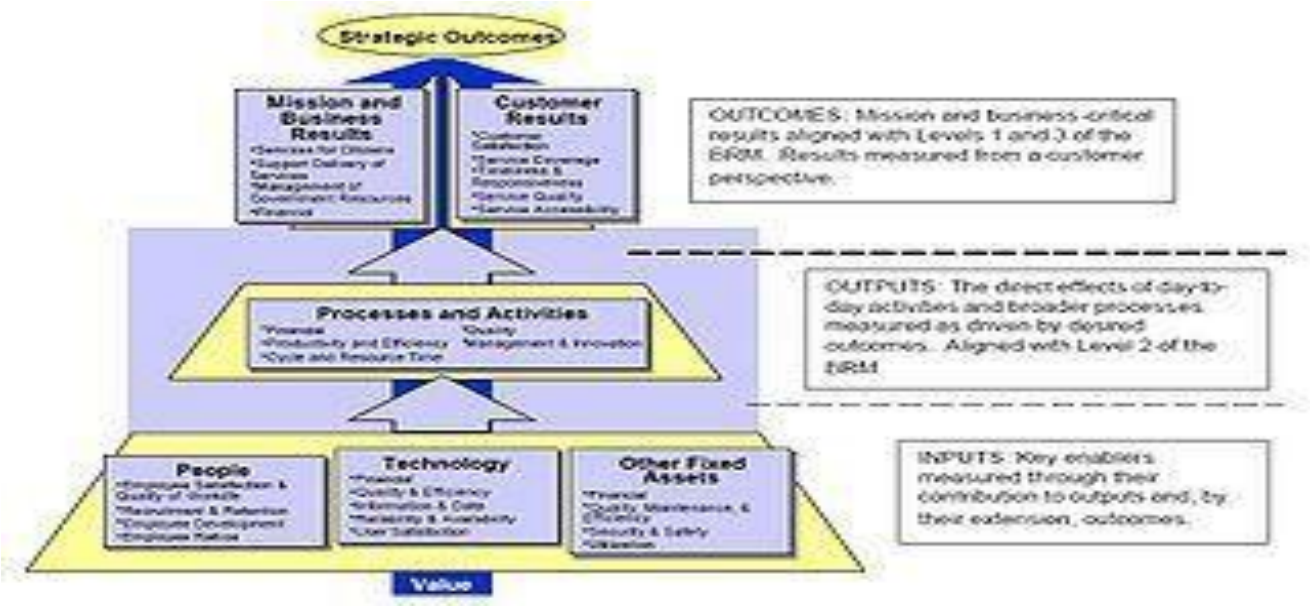
Provide a common language for communication

Are explicitly defined in terms of owner, unit of measure, collection frequency, data quality, expected value(targets), and thresholds

Are valid, to ensure measurement of the right things

Are verifiable, to ensure data collection accuracy

EX-



Performance Reference Model of the Federal Enterprise Architecture,

This process of measuring performance often requires the use of statistical evidence to determine progress toward specific defined organizational objectives. Performance measurement is a fundamental building block of TQM and a total quality organization.

Historically, organisations have always measured performance in some way through the financial performance, be this success by profit or failure through liquidation. However, traditional performance measures, based on cost accounting information, provide little to support organizations on their quality journey, because they do not map process performance and improvements seen by the customer. In a successful total quality organisation, performance will be measured by the improvements seen by the customer as well as by the results delivered to other stakeholders, such as the shareholders.

This section covers why measuring performance is important. This is followed by a description of cost of quality measurement, which has been used for many years to drive improvement activities and raise awareness of the effect of quality problems in an organisation.

A simple performance measurement framework is outlined, which includes more than just measuring, but also defining and understanding metrics, collecting and analysing data, then prioritising and taking improvement actions. A description of the balanced scorecard approach is also covered.

Why measure performance?

„When you can measure what you are speaking about and express it in numbers, you know something about it".

„You cannot manage what you cannot measure".

These are two often-quoted statements that demonstrate why measurement is important. Yet it is surprising that organisations find the area of measurement so difficult to manage.

In the cycle of never-ending improvement, performance measurement plays an important role in:

• Identifying and tracking progress against organisational goals

• Identifying opportunities for improvement

• Comparing performance against both internal and external standards

Reviewing the performance of an organisation is also an important step when formulating the direction of the strategic activities. It is important to know where the strengths and weaknesses of the organisation lie, and as part of the *„Plan –Do – Check – Act"* cycle, measurement plays a key role in quality and productivity improvement activities. The main reasons it is needed are:

• To ensure customer requirements *have* been met

• To be able to set sensible *objectives* and comply with them

• To provide *standards* for establishing comparisons

• To provide *visibility* and a ―scoreboard‖ for people to *monitor* their own performance level

• To highlight *quality problems* and determine areas for *priority attention*

• To provide *feedback* for driving the improvement effort

It is also important to understand the impact of TQM on improvements in business performance, on sustaining current performance and reducing any possible decline in performance.

1.6 RECOMMENDED QUESTIONS

1. Define Data Structures?

2.What is a pointer variable?

3. Difference between Abstract Data Type, Data Type and Data Structure?

4. Define an Abstract Data Type (ADT)?

5. Give any 2 advantages and disadvantages of using pointers?