**Data structures and Algorithms**

**Unit 1: Concept and Definition of Data Structure**

**(4hrs, 5 marks)**

**1.1 Information and its meaning**

Computer science is fundamentally the study of information. Computer manipulates data into information in binary format. In binary format, data are represented either by 1's or by 0's. Such individual digit is called bit. 1 is also used to represent high or true whereas 0 is used to represent low or false. In computer system 1's or 0's are represented by 5 volt and 0 volt respectively. Collection of such raw facts 1's and 0's are called data. Data independently has no meaning itself. It should be processed further to make it meaningful. The output of processed data is called Information. Data are represented in various ways which of them are as follows.

**Binary number system**

The most widely used method for interpreting bit settings as nonnegative integers is the binary number system. Each bit position represents a power of 2 . For example, 00100101 has 1's in position 0 , 2 and 5. Thus 00100101 represents the integer 20 + 22 + 25 = 1 + 4 + 32 = 37. The n length number represents a unique non negative integer between range 0 and 2n-1. For ex 23 = 8, to represent eight digits uniquely we need 3 bits.

To represent negative numbers, there are two techniques

**Ones complement notation**

a negative number is represented by changing each bit in its absolute value to the opposite bit setting. For example, since 00100101 represents 37 , 11011010 is used to represents – 37. MSB is used to store sign of the number.

If MSB is 0, number is positive else negative

**Twos complement notation**

1 is added to the ones complement representation of negative numbers to get twos complement. For example, since 00100101 represents 37, 11011010 is used to represents – 37. 1101100011 is used to represent twos complement representation. Its doesn't yield negative zero.

**Real number**

Real numbers are represented in floating point notation

**Character string**

Name, Job,Title etc can't be represented numerically. Such information is represented in character string form. 8 bits are required to represent 256 different characters. The number of bits necessary to represent a character in a particular computer is called the byte size and the group of bits of that number is called a byte.

We generally interpret bit pattern to give its meaning. For example, the bit string 00100110 can be interpreted as the number 38 (binary), the number 26 (binary decimal coded) or the character '&'. The method of interpreting a bit pattern is often called a **data type.** Such as integer, float, character, real number

A type is a collection of values, like

**Z** = {..., ­2, ­1, 0, 1, 2, ...}

**N** = {0, 1, 2, 3, 4, ...}

**B** = {false, true}

**C** = the set of characters

**R** = the set of all real numbers A data type is a type together with operations.

**Hardware and software**

We store information in computer memory. Computer memory are simply a group of bits. At any instance, the bit of a computer memory are either 0 or 1. i.e (off or on) The combination of 0's and 1's stored in memory are called its value or content.The group of bits that store in the computer memory are called **words.** The words may be of different type according to their assignment

Control word : stores control information of computer such as status, flag etc.

Address word: stores address of data

Data word: Stores actual data represented by address word

**Why data structures?**

Ultimate goal ­­- to write efficient programs. In order to do that, one needs to organize the data in such a way that it can be accessed and manipulated efficiently. Data structures help us to organize the data in the computer, resulting in more efficient programs. An efficient program executes faster and helps minimize the usage of resources like memory, disk.

What actually **data structure** is

It’s an agreement about:

* how to store a collection of objects in memory,
* what operations we can perform on that data,
* the algorithms for those operations, and
* how time and space efficient those algorithms are.

Ex.Array in C:

Stores objects sequentially in memory

Can access, change, insert or delete objects

Algorithms for insert & delete will shift items as needed

Space: O(n), Access/change = O(1), Insert/delete = O(n)

**What are data structures?**

There are many definitions available:

A data structure is a way of arranging data in a computer's memory or other disk storage.

A data structure is a collection of data, organized so that items can be stored and retrieved by some fixed techniques.

**What is an algorithm?**

Informally, an algorithm is any well defined computational procedure that takes some value or set of values, as **input** and produces some value, or set of values; as **output.** An algorithm is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a **tool** for solving a well defined-specified computational problem.

There are many definition of algorithms:

An algorithm is a procedure, a finite set of well defined instructions, for solving a problem which, given an initial state, will terminate in a defined end state. The computational complexity and efficient implementation of the algorithm are important in computing, and this depends on suitable data structures.

An algorithm is said to be correct if, for every input instance, it halts with the correct output. We say that a **correct** algorithm solves the given computational problem.

**Theory Assignment 1.1**: What kinds of problems are solved by algorithms?

**1.2.Array in C**

An array is a data structure which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same types. Instead of declaring individual variables, such as number0, number1,....., number99. You can declare variable such as numbers and use number[0] , number[1] , number [2] and -----number[99] to represent individual variables. A specific element in an array is accessed by an index.

All array consist of contiguous memory location. The lowest element corresponds to the first element and the highest address to the last element.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Number[0]  (first element) | Number[1] | Number[2] | Number[3] | Number[n]  (last element) |

**Declaring Arrays**

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:

*type arrayName [ arraySize ];*

This is called a single - dimensional array. The array size must be an integer constant greater than zero and type can be any valid C data type. For example to declare a 10 – element array called balance of type double , use following statement:

*double balance [ 10 ] ;*

now balance is a variable array which is sufficient to hold up to 10 double numbers.

**Initializing Arrays**

You can initialize array in C either one by one or using a single element as follows:

*double balance [ 5 ] = { 1000.0, 2.0 , 3.4, 7.o, 50.0};*

The number of values between braces{} can not be larger than the number of elements that we declare for the array between square brackets [].

Or you can omit the size, the required size will created during initialization as follows:

*double balance [ ] = { 1000.0, 2.0 , 3.4, 7.o, 50.0};*

Exactly the same array is created as you did in the previous example. Following is an element to assign a single element of the array

*balance [ 4] = 1000.0;*

The above statement assign element number 5th in the array with a value of 100.

**Accessing Array**

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example.

*Double salary = balance [4];*

The above statement will take fifth element from the array and assign the value to salary variable.

1.3. **The Array as an ADT**

A useful tool for specifying the logical properties of a data type is the ***abstract data type, or ADT.*** Fundamentally a data type is a collection of values and a set of operations on those values.That collection and those operation from a mathematical construct that may be implemented using a particular hardware or software data structure. The term “abstract data type” refer to the basic mathematical concept that defines the data type. An abstract data type is a data type that instead of specifying implementation details such as memory storage requirements, memory layout, specifies only the behavior of some conceptual structure.Space and time are implementation issue and we are not considered those issues here. By specifying the mathematical and logical properties of a data type or structure, the ADT is a useful guideline to implementors and a useful tool to programmers who wish to use the data type correctly.

There are numbers of methods for specifying an ADT. The method we use here is based on C. Lets take an example of rational number.

In mathematics, a **rational number** is any number that can be expressed as the quotient or fraction *p*/*q* of two integers, with the denominator *q* not equal to zero.

*/\** ***value definition*** *\*/*

***abstract typedef*** *< integer, integer> RATIONAL // Definition Clause: define Rational no*

***condition*** *RATIONAL [1] ! = 0; //Condition Clause: denominator should not be zero*

*//Conditional Clause is Optional*

*/\** ***operator definition*** *\*/*

***abstract*** *RATIONAL makerational(a,b) //Header*

*int a,b; // Header*

***precondition*** *b!=0; // precondition required*

***postcondition*** *makerational[0] = a; // postcondition*

*makerational [1] = b;*

***abstract*** *RATIONAL add(a,b) /\* Written a+b\*/*

*RATIONAL a , b;*

***postcondition*** *add[1] ==a[1]\*b[1];*

*add[0] == a[0] \* b[1] + b[0] \* a [1];*

***abstract*** *RATIONAL mult (a,b) /\* Written a \* b \*/*

*RATIONAL a , b;*

***postcondition*** *mult[0] == a[0] \* b[0];*

*mult[1] == a[1] \* b[1];*

***abstract*** *RATIONAL equal (a,b) /\* Written a == b\*/*

*RATIONAL a , b;*

***postcondition*** *equal == (a[0] \* b[1]== b[0] \* a[1];*

An ADT consist of two parts: a value definition and an operator definition. The value definition defines the collection of values for the ADT and consist of two parts: a definition clause and condition clause. The keywords abstract typedef introduce a value definition, and the keyword condition is used to specify any condition on the newly defined type.Conditional part is optional.

Immediately, value definition is followed by operator definition. Each operator is defined as an abstract function with three parts:

* a header
* the optional precondition
* and the postcondition

The first two line above example inside operator definition are header. For example:

***abstract*** *RATIONAL add(a,b) /\* Written a+b\*/*

*RATIONAL a , b;*

These are header portion of operator definition. Abstract indicates that function are not exactly implented as in c but an ADT operator definition. Alternatively header can be written using keyword written as shown in above comment line

The precondition checks the condition that should be met before executing next step. For example:

***precondition*** *b!=0;*

Here, the number to be a rational, it should not be divisible by 0. It is checked in precondition as shown above. The postcondition will be executed after precondition guarantees that the numerator is not equal to zero. It is an optional step and not necessary to initialise all the time.

The actual operation that should be carried on the data are defined in postcondition. The name of the function such as *add , mult* is used to store the result of the operation.Thus mult [0] represents the numerator of the result, the mult[1] the denominator of the result.

For multiplication,

\*=

For Addition,

+=

We can also represent array as an ADT as follows:

***abstract typedef*** *<< eltype, ub >> ARRTYPE(ub, eltype*);

***condition*** *type(ub) = = int;*

***abstract*** *eltype extract( a, i) /\* written a[i] \*/*

*ARRTYPE(ub, eltype) a;*

***int*** *i;*

***precondition*** *0<= i < ub;*

***postcondition*** *extract == ai*

***abstract*** *store(a, i, elt) /\* written a[i] =elt \*/*

*ARRTYPE ( ub, eltype) a;*

***int*** *i;*

*eltype elt;*

***precondition*** *0 <= i < ub;*

***postcondition*** *a[i] == elt;*

Let *ARRTYPE ( ub, eltype)* denote the ADT corresponding to the C arraytype *eltype array[ub] .* Here*, ub* and *eltype* are the parameters which represents upper bound and element type respectively. ARRTYPE (10 , int) would represent the type of the array x in the declaration int x[10].

From above discussion, simply array as an ADT in high level language are decleread in following ways:

Let us suppose , We have an array ***A*** of type ***ARRYTYPE*** has n element defined by upper bound ub then following primitive operation can be performed:

CREATE ( A) : create an array A.

INSERT( A, ELT) : Insert element ELT on Array A

DELETE(A, ELT) : Delete element ELT from Array A

MODIFY(A, OLDELT, NEWELT) : Modify old element OLDELT by new element NEWELT

TRAVERSE(A): Find all the elements of array A

1.4. **1D Array**

A one dimensional array is used when it is necessary to keep a large number of items in memory and reference. The first starting address is also called base address and is denoted as base(b). Suppose size of each individual element is *arraySize.* To access item on array[i], we calculate as *base(b) + i \* arraySize*

Declaring, Initializing and Accessing of 1D array are explained in section 1.3

***Lab 1: Read 100 integers, find their average, and determine by how much each integer deviates from that average.***

**1.5. 2D Array**

2D array is used to store ,the tabular representation of, table data. A two-dimensional array is really nothing more than an array of arrays (a three-dimensional array is an array of arrays of arrays).

11 12 13

A= 14 15 16

17 18 19

The above mentioned matrix is 3 by 3. The matrix elements can be accessed using the formula A[m,n], where m represents row number and n represents column number.

Thus, the first element i.e. 11 is represented as A[0,0].

Similarly,

A[0,1] = 12

A[0,2] = 13

A[1,0] = 14

A[1,1] = 15 and so on.

**Initialization of 2D Array**

The 2-dimensional array follows the similar concept. So we can declare 2-dimensional array for above matrix as A[3][3].

We can **define 2-dimensional array** as follows.

int A[3][3]={11,12,13,14,15,16,17,18,19}

Element 11 can be referred as A[0][0]

Element 12 can be referred as A[0][1]

Element 13 can be referred as A[0][2]

Element 14 can be referred as A[1][0]

Element 15 can be referred as A[1][1] and so on.

Another way to define 2-D array is:

int A[3][3]={

{11,12,13},

{14,15,16},

{17,18,19}

}

The above mentioned method increases the readability of the matrix.

int A[][] and A[3][] are invalid. We cannot skip the column index in 2-D arrays.

int A[][3] and A[3][3] are both valid.

**Accessing 2D Array**

Any element of 1D array can be accessed by using a for loop. But in 2D, we need two nested to access any element. For example the element at A[i][j] is accessed when first loop is at i and second is at j. i.e A[2][3] is accessed when i=2 and j=3;

**Application of 2D Array**

* Storing a table of data (not the only way).
* Any kind of matrix processing, as a 2D array really is a matrix.
* Modeling relationships, networks, and maps with a graph.

**1.6.**  **Multi D Array**

Assignment:

**1.7.** **Structure**

The structure is a collection of data items of different types using a single name. A structure is a convenient tool for handling a group of logically related data items. For example it can be used to represent a set of attributes such as student\_name, roll\_number and marks. The general form of a structure definition is as follows:

Struct tag\_name

{

Data type member 1;

Data type member 2;

------------ ------------- ;

------------ ------------- ;

};

For example, if we have, variable book\_store, then declaration of structure variable book\_store is given as,

Struct book\_store {

char title [20];

char author [10];

int page;

float price;

}; struct book\_store book 1, book 2, book 3;

**1.8. Union**

Unions are a concept borrowed from structures and therefore follow the same syntax as structures. However, there is major distinction between them in terms of storage. In structures, each member has its own storage location, whereas all the members of a union use the same location. A union can be declared using the keyword union as follows ;

Union item

{

int m;

float x; members of union.

char c;

}code;

1.9.  **pointer**

A pointer is nothing but a variable that contains an address of a location in memory. We can declare the variable ptr as a pointer to an integer as

int \*ptr;

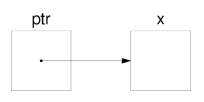
You can think of this as being a new type, which is (int \*), which means ”pointer to an integer.” When it is declared, a pointer does not have a value, which means that it does not yet store an address in memory, and hence it does not point to an address in memory. Let’s say we want to have the pointer that we deﬁned point to the address of another integer. This is done with

int x;

int \*ptr;

ptr = &x;

The unary operator & returns the address of a particular variable. We can represent the above three lines of code graphically with

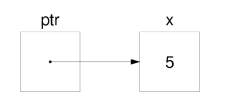


Since we did not assign the value of x, it does not contain a value. We can assign the value

of x in one of two ways. The ﬁrst way is the most obvious way, which is simply

x=5;

The pointer diagram now has a value at the location in which the variable x is stored:



The other way we can assign the value of x is to use the pointer that points to its address.

Since ptr points to its address, we can assign a value to the address that this pointer points

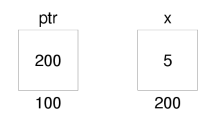
to with

\*ptr = 5;

This is called dereferencing the pointer and it is identical to x=5;. The dereferencing operator

says to set the contents of location pointed to by the pointer to the speciﬁed value.

Pointer is used to implement indirect addressing. The memory addresses which stores actual operand are called effective address. The direct address stores effective address. Indirect addressing holds address of direct addressing, i.e address of address, where actual operand are stored.



Here, memory location 100 stores the address of actual operand i.e 200 . Here the actual operand is 5 and effective address is 200. So, memory location 100 is indirect addressing and holds the effective address 200 where actual operand 5 is stored. This condition in memory addressing is implemented by using pointer.

**Unit 2: Algorithm**

**(2 hrs and contains 3 marks)**

1. **Concept and Definition:**

Informally, an algorithm is any well defined computational procedure that takes some value or set of values, as **input** and produces some value, or set of values; as **output.** An algorithm is is thus a sequence of computational steps that transform the input into the output.

We can also view an algorithm as a **tool** for solving a well defined-specified computational problem.

There are many definition of algorithms:

* An algorithm is a procedure, a finite set of well defined instructions, for solving a problem which, given an initial state, will terminate in a defined end state. The computational complexity and efficient implementation of the algorithm are important in computing, and this depends on suitable data structures.
* An algorithm is said to be correct if, for every input instance, it halts with the correct output. We say that a **correct** algorithm solves the given computational problem.

1. **Design of Algorithm**

Algorithm design is a specific method to create a mathematical process in solving problems. Applied algorithm design is [algorithm engineering](http://en.wikipedia.org/wiki/Algorithm_engineering).

Algorithm design is identified and incorporated into many solution theories of [operation research](http://en.wikipedia.org/wiki/Operation_research), such as [dynamic programming](http://en.wikipedia.org/wiki/Dynamic_programming) and [divide-and-conquer](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm). Techniques for designing and implementing algorithm designs are algorithm design patterns,[[1]](http://en.wikipedia.org/wiki/Algorithm_design#cite_note-1) such as [template method pattern](http://en.wikipedia.org/wiki/Template_method_pattern) and [decorator pattern](http://en.wikipedia.org/wiki/Decorator_pattern), and uses of data structures, and name and sort lists. Some current day uses of algorithm design can be found in internet retrieval processes of web crawling, packet routing and caching.

Mainframe programming languages such as [ALGOL](http://en.wikipedia.org/wiki/ALGOL) (for *Algo*rithmic *l*anguage), [FORTRAN](http://en.wikipedia.org/wiki/FORTRAN), [COBOL](http://en.wikipedia.org/wiki/COBOL), PL/I, [SAIL](http://en.wikipedia.org/wiki/SAIL_programming_language), and SNOBOL are computing tools to implement an "algorithm design"... but, an "algorithm design" (a/d) is not a language. An a/d can be a handwritten process, e.g. set of equations, a series of mechanical processes done by hand, an analog piece of equipment, or a digital process and/or processor.

One of the most important aspects of algorithm design is creating an algorithm that has an efficient run time, also known as its [big Oh](http://en.wikipedia.org/wiki/Big_Oh).

Steps in development of Algorithms

1. Problem definition
2. Development of a model
3. Specification of Algorithm
4. Designing an Algorithm
5. Checking the correctness of Algorithm
6. Analysis of Algorithm
7. Implementation of Algorithm
8. Program testing
9. Documentation Preparation

Basically there are three types of algorithm design technique.

* **Divide and conquer**

Divide and Conquer algorithms break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

There are three steps to applying Divide and Conquer algorithm in practice:

* **Divide** the problem into one or more subproblems.
* **Conquer** subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
* **Combine** the solutions to the subproblems into the solution for the original problem
* **Dynamic programming**

Dynamic programming is a design technique similar to divide and conquer. Divide-and-conquer algorithms partition the problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem. Dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems. A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subsubproblem is encountered.

There are two ways of doing this.

**1.) Top-Down :** Start solving the given problem by breaking it down. If you see that the problem has been solved already, then just return the saved answer. If it has not been solved, solve it and save the answer. This is usually easy to think of and very intuitive. This is referred to as ***Memoization***.

**2.) Bottom-Up** **:** Analyze the problem and see the order in which the sub-problems are solved and start solving from the trivial subproblem, up towards the given problem. In this process, it is guaranteed that the subproblems are solved before solving the problem. This is referred to as Dynamic Programming.

Note that divide and conquer is slightly a different technique. In that, we divide the problem in to non-overlapping subproblems and solve them independently, like in mergesort and quick sort.

Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution *an optimal solution*, as opposed to *the optimal solution*, since there may be several solutions that achieve the optimal value.

Dynamic programming can be effectively applied to solve the longest common subsequence (LCS) problem. The problem is stated as following: given two sequences (or strings) x and y find a maximum-length common subsequence (substring) of x and y.

For example, given two sequences x = "ABCBDAB" and y = "BDCABA", the LCS(x, y) = { "BCBA", "BDAB", "BCAB" }. As you can see there are several optimal solutions.

* **Greedy Paradigm**

A greedy algorithm repeatedly executes a procedure which tries to maximize the return based on examining local conditions, with the hope that the outcome will lead to a desired outcome for the global problem. In some cases such a strategy is guaranteed to offer optimal solutions, and in some other cases it may provide a compromise that produces acceptable approximations.

1. **Characteristics of algorithm**

While designing an algorithm as a solution to a given problem, we must take care of the following five important characteristics of an algorithm.

**Finiteness:**

An algorithm must terminate after a finite number of steps and further each step must be executable in finite amount of time. In order to establish a sequence of steps as an algorithm, it should be established

that it terminates (in finite number of steps) on all allowed inputs.

**Definiteness (no ambiguity):**

Each steps of an algorithm must be precisely defined; the action to be carried out must be rigorously and unambiguously specified for each case.

**Inputs:**

An algorithm has zero or more but only finite, number of inputs.

**Output:**

An algorithm has one or more outputs. The requirement of at least one output is obviously essential, because, otherwise we cannot know the answer/ solution provided by the algorithm. The outputs have specific relation to the inputs, where the relation is defined by the algorithm.

**Effectiveness:**

An algorithm should be effective. This means that each of the operation to be performed in an algorithm must be sufficiently basic that it can, in principle, be done exactly and in a finite length of time, by person using pencil and paper. It may be noted that the ‘FINITENESS’ condition is a special case of ‘EFFECTIVENESS’. If a sequence of steps is not finite, then it cannot be effective also.

1. **“Big - Oh” Notation**

**Time Complexity And Space Complexity**

Time complexity is an analysis of the amount of time required to solve a problem of a particular size,while the Space complexity is an analysis of the amount of memory required.

**Analysis Of Algorithm**

The theoretical study of computer program performance and resource usage

**What more important than Performance?**

Correctness, stability, features, functionality, security, user friendliness, modularity, Simplicity, reliability

**Why study algorithm and performance?**

Scalability

Feasible vs Infeasible

performance is like money

JAVA and c

**Kinds of analysis**

**Worst Case (usually)**

T(n) = max time on any input of size n

**Average Case(sometimes)**

T(n)=expected time over all input of size n

**(Need assumption of statistical distribution)**

**Best Case Analysis:(bogus: not good)**

Cheat with a slow algorithm that works fast on some input

**What is insertion sorts W-c time?**

It depends on the speed of our computer.

* absolute speed (on diff machine)
* relative speed(on same machine)

**BIG IDEA: Asymptotic analysis**

1. Ignore machine dependent constant
2. Look at growth of T(n) as n->

**Asymptotic notation**

**ʘ notation:**

1. Drop low order terms
2. Ignore Leading constants

**EX** 3n³+90n²+10n=ʘ(n³)

**As** n->**ʘ(n²) always beats ʘ(n³)**

**what do you mean by Asymptotic :**

A line that continually approaches a given curve but does not meet at any finite distance

**Asymptotic Notation:**

Asymptotic Notation primarily deal with running time of algorithm.

***O* notation (Upper Bound):**

f(n)=*O*(g(n)) means there are constants C>0, no>0 such that 0 < = f(n) < c \* g(n) for all n = n0

**Ex. 2n2 = *O* (n3) means 2n2 *O* (n3)**

**Set definition**

For a given function g(n) , we denote by(g(n)) the set of function

*O*(g(n))={ f(n): there exist positive constants c>, no such that 0 < = f(n) <= c \* g(n) for all n=n0

}

**Data structures and Algorithms**

**Theory Assignment: 1**

**Assign Date: 14th, June Due Date: 24th, June**

1. What is Data type? Define, data structure with with examples.
2. What is Abstract Data Type?
3. What is the difference between ADT and data structure?
4. What is an algorithm? What are the characteristics of algorithm? Explain.
5. What do you mean by complexity of Algorithm? Explain.
6. What is best case, average case and worst case ? Which case is used more and why?
7. What is Big Oh (O) notation?
8. Compare Big Oh, Big Omega and Big theta notation.
9. What are the different algorithm design techniques? Explain.
10. What is Dynamic programming?