First Semester MCA - CBCS Y2K20 Scheme

**1MCA6 - Data Structures**Reference Document

Unit - I: Introduction and Overview

**\*\*\*Part-1 of Unit-I\*\*\***

**Basic Terminologies: Elementary Data Organization**

Data is set of values, when processed it produces information. A processed information can be represented using a pair of attributes and values, which in turn called as an entity. For example:

Attributes: Name Age Address

Values: Ajay 30 Kengeri, Bangalore

Such attributes contribute towards formation of records. And collection of records is called files.

**Data Structures**

An efficient method to store, retrieve and manage data is called data structures. It is a logical or a mathematical model of organization of data. Every data structure must:

1. Have a rich structure to reflect the relations between data in real-world
2. Be simple enough to allow the operations on data to be done with ease

**Classification of data**

Following are the classification of data structures based on their nature,

S Primitive Data Structures

* Integer - 10, 3, 1020 etc.
* Real - 5.5, 96.77, 10.0 etc.
* Character - 'a', 'b', '2' etc.
* Boolean - True or False, 0 or 1 etc.

S Non-primitive data structures

o Linear Data Structures: collection of elements in a sequence

* Arrays
* Linked list
* Stacks
* Queues

o Non-linear Data Structures: non-sequential collections

* Trees
* Graphs

**Arrays**

Arrays are a sequential collection of a finite number of elements having similar data type. Example:

A1, A2, A3, An or {1, 2, 3, 4, .... n} etc.

Some characteristics of arrays:

S Array elements are sequentially processed.

S Elements can be randomly accessed using indices. For example: A[3] would access 4th element in an array A.

**Linked Lists**

An ordered collection of data elements with each element having a link to another element. Linked lists have elements that can be divided into two parts, data and link. Following image shows a visual representation of linked lists.



Fig 1.1 - Each cell contains a data and a link. A link points to the next cell. **Note:** Linked lists can be circular or two-sided depending on the definition. **Trees**

Trees in data structures form a hierarchical representation of data. Traversing through the parts of the tree begins at root element. Trees can be very useful in data representation. For example, an equation 2x2+y can be represented in a tree form as,

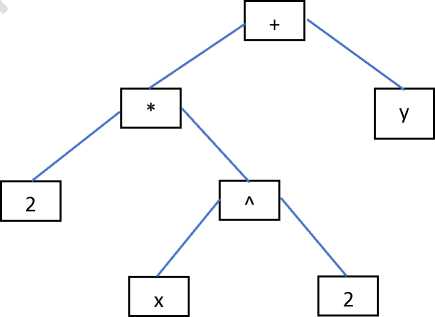


Fig 1.2 - A tree representing the equation 2x2+y

**Other data structures**

Data structures offer many ways of storing and retrieving data. Stacks, queues and graphs are few to mention. All these data structures are built to store, retrieve, delete and manage data in an efficient manner based on the given scenario. Details are furnished in the subsequent units.

**Data structure operations**

There are four basic operations on any data structures:

1. Traversing: access the data records
2. Searching: finding the location of a data record
3. Inserting: adding a new data record
4. Deleting: removing a data record

Apart from the basic operations, data structures can be operated with some special functions,

1. Sorting: arranging the data records in some logical order
2. Merging: combining the multiple data records into one

**Abstract Data Types (ADT)**

Implementation of a function can be hidden to reduce the complexity of an operation. When a set of data values and associated operations are defined in such a way that the implementation is hidden, they are called ADT.

Each of the data structure can have its own ADT. For example, String ADT, Stack ADT, Linked List ADT etc.

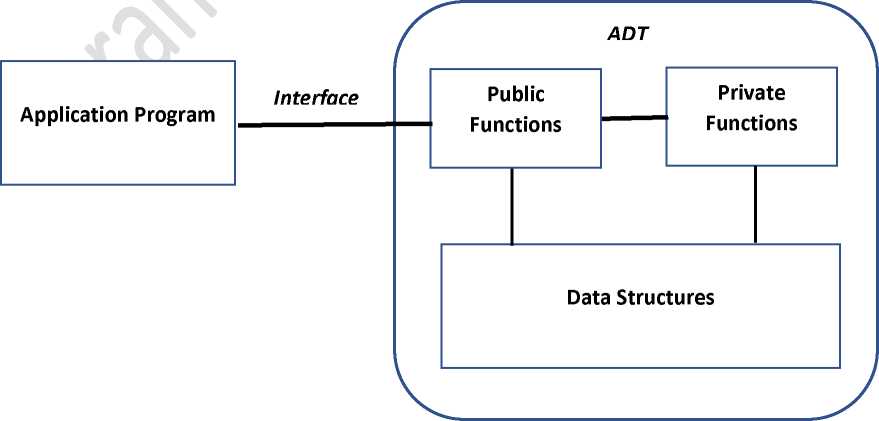


Fig 1.3 - An abstract data type model

**Mathematical Notations and Functions**

Following are the description on various mathematical notations used while writing the algorithms on operations of data structures.

1. Floor and ceiling functions

Any real number x falls between two integers called the floor and the ceiling of x.

* floor(x) is used to denote floor value of x, which is the greatest integer that does not exceed x.
* ceiling(x) is used to denote ceiling value of x, which is the smallest integer that is not less than x.

For example:

floor(5.6) = 5 , floor(10.2) = 10, floor(V5) = 2, floor(-3.6) = -4 etc.. ceiling(6.6) = 7, ceiling(-8.2) = -8, ceiling(10) = 10 etc..

1. Remainder function: Modular arithmetic

r = k mod M (k modulo M), r represents a remainder integer number when k is divided by M.

Example: 12 mod 10 = 2, 19 mod 5 = 4, 809 mod 100 = 9 etc.

The term "mod" is also used for the mathematical congruence relation, which is denoted as:

a = b (mod M) if and only if M divides b-a here M is called the modulus.

1. Integer and absolute value functions

If x is a real number then, INT(x) converts x into an integer value after removing the fraction part.

For example:

INT(3.14) = 3, INT(V5) = 2, INT(-8.5) = -8, INT(7) = 7

The absolute value of a real number x, written ABS(x) or |x| is defined as the greater of x or -x.

For example:

|0| = 0, |-3.33| = 3.33, |-10| = 10, | -0.089| = 0.089, |8| = 8

1. Summation

A summation operation represented by X (the Greek letter sigma). When

applied on a sequence a1, a2, a3, an summation would sum all the values

and return the sum of the sequence.

For example:

E?=0 ai = ai+a2+%+ +an

1. Factorial Function

The product of positive integers from 1 to n is denoted as n! (or n factorial). It can be represented as,

n! = 1 \* 2 \* 3 \* \* (n-1) \* n

For example:

5! = 120, 2! = 2, 0! = 1

1. Permutation

A permutation of a set of n elements is an arrangement of the elements in a given order. For example, the permutation of the set consisting elements a, b and c are:

abc, acb, bac, bca, cab, cba

For a set consisting of n elements, n! number of permutations can be derived.

1. Exponents and Logarithms

An exponent of an integer number represented by xm is the multiplication of the number x itself m number of times.

For example:

xm = x \* x \* x \* \* x (m times)

52 = 5 \* 5 = 25, 50 = 1, 5-2 = 1 / 52

Logarithms are related to exponents as follows. Let b be a positive number. The logarithm of any number x to the base b is written as, logb x

represents the exponent to which b must be raised to obtain x. That is, if logb x = y then by = x

For example,

log2 8 = 3 since 23 = 8

**Algorithms: Complexity and time-space trade-off**

Efficiency of an algorithms is measured based on the time and space it consumes for a provided solution.

Time complexity of an algorithm is represented in terms of n (the input size). For example, a binary search would have log2 n as the worst case time complexity.

Meaning, a binary search on a list of 25,000 sorted elements would require not more than 15 comparisons.

On the other hand, a linear search would require n number of comparisons in the worst case.

An algorithm can be made efficient by trading time / space. For example, a file containing thousands of names can be sorted to apply binary search. That would make searching efficient. However, sorting requires additional space as the elements need to be stored temporarily to a new location while sorting. This concept of trading time for space or vice versa is called time-space trade-off.

**Algorithmic Notations**

An algorithm is a finite step-by-step list of well-defined instructions to solve a problem. Following is an example of an algorithm to find the location of a maximum value stored in an integer array:

Largest\_of\_the\_array(integer data) // input is an integer array named data Stepl. Initialize, n as the size of array, k := 1, loc := 1 and max := data[1]

Step2. Set k := k + 1

Step3. If k > n, then: write loc, max and Exit. // when the array is fully explored Step4. If max < data[k], then set loc := k and max := data[k]

Step5. Go to Step2

Following is the C implementation of the above algorithm:

#include<stdio.h> void main(){

int data[10] = {1,2,3,4,5,98,33,6,0}; int n=9, k, loc, max = data[0]; for(k=1;k<=n;k++){ if(k==n)

{

printf("Location=%d, Maximum=%d",loc,max); break;

}

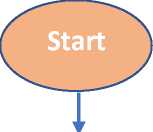
if(max<data[k]){ loc = k; max = data[k];

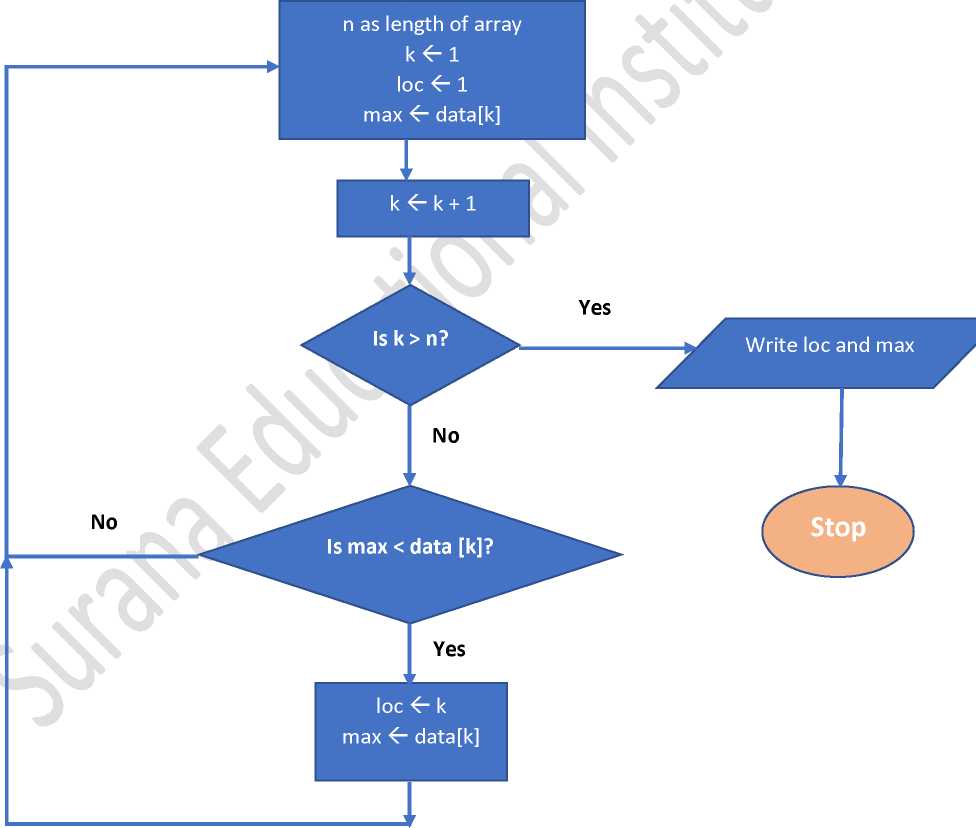
}

}

}

Following is the flowchart for the above algorithm:



Following are the category of notations used in an algorithm design:

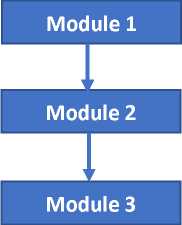
1. Comments: used to describe the instructions
2. Input and output: at least one output and zero or more inputs are taken
3. Variables: used to represent the data values
4. Assignment statements: used to assign values to variables
5. Steps, control, exit: instructions through which the algorithm flows
6. Procedures: a sub-algorithm that solves part of the problem

**Control Structures**

An algorithm can make use of following types of flow of control through the modules while providing solution to a problem:

1. Sequence logic or sequence flow

When the instructions are executed in an obvious sequence, the type of logic is termed as sequenced flow. This type of flow is elementary in an algorithm design. Following diagram shows a sequential flow:



1. Selection logic or conditional flow

A selection logic includes a condition that gets evaluated and results in selection of one among several choices of modules. Further, a selection logic can be divided into three types:

1. Single alternative : if (condition) then

module End if

1. Double alternative : if (condition)

module-1

else

module-2

1. Multiple alternatives : if (condition-1)

module-1

else if (condition-2) module-2 else if

1. Iteration logic or repetitive flow

Iteration logic implements the execution of modules n number of times. Multiple finite iterations are done on a set of instructions.

For example:

Repeat module-1 to module-3 for k=1 to n: module-1 module-2 module-3 End iteration

**Complexity of algorithms**

Complexity of an algorithm is denoted by a function f(n), where n is the input size of an algorithm. The function gives the running time and/or storage space required by the algorithm. Complexity of algorithms are usually represented as one of the following cases:

1. Worst case: the maximum value of f(n)
2. Average case: the expected value of f(n)

Asymptotic notations are used to represent the complexity of algorithms. Big oh (O), Theta (0) and Omega (Q) are some renowned asymptotic notations used to represent the time complexity of algorithms in worst case, average case and best case respectively. For example, here are few complexity representations of known algorithms:

1. Linear search: O(n)
2. Binary search: O(log n)
3. Bubble sort: O(n2)
4. Merge-sort: O(n log n)

**\*\*\*End of Part-1 of Unit-I\*\*\***