Data Structures & Program Design

Introduction

Prof. D. A. Borikar
Department of Computer Sc. & Engg.

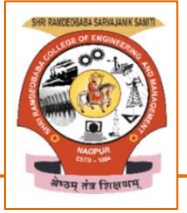
DATA STRUCTURES

A data structure is a way to store and organize data in order to facilitate access and modifications.

Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways.

The basic motivations to study data structure are -

- To identify and create usable mathematical entities and operations for the problems which requires a solution, and
- To determine the representation of the entities and the implementation of the operations on such problems.



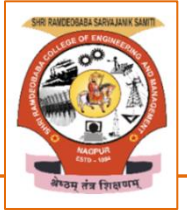
A data structure represents the knowledge (understanding and interpretation) of data to be organized in the computer memory. It should be designed and implemented with a view to reduce the complexity and increase the efficiency.

This essentially involves data abstraction, problem analysis and algorithm design.

The data structures may be classified as – arrays, lists and files.

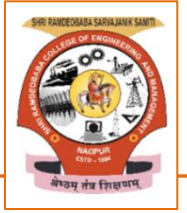
Further, the lists can be categorized into –

- linear lists (i.e., stacks, queues, linked lists) and
- non-linear lists (i.e., trees, graphs).



Data structures that stores elements of same type are homogenous data structures, while those storing elements of different types (i.e., the records defined using **struct** and **union** in C) are heterogeneous data structures.

Generally the primitive data structures includes integers, real numbers, characters and pointers; whereas the non-primitive data structures includes arrays, lists and files.



Data structures are also categorized on the basis of allocation of memory by the operating system (i.e., at compile time or at execution time).

The structures for which the storage is fixed and allocated at compile time are static structures (for example: arrays).

The structures for which the storage is variable and allocated during execution time are dynamic structures (for example: linked lists using pointers).

CLASSIC STRUCTURES

ARRAYS

An array is an ordered set which consists of a fixed number of objects.

No deletion or insertion operations are performed arrays.

At best, elements can be changed to a value which represents an element to be ignored (setting an element in an array to zero to delete it).

LISTS

A list is an ordered set consisting of a variable number of elements to which insertions and deletions can be made.

A list which displays the relationship of adjacency between elements is said to be linear. Any other list is said to be nonlinear.

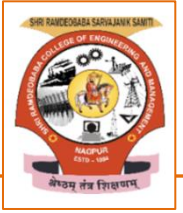
Operations performed on lists include those which are performed on arrays. The important difference is that the size of a list may be changed by updating (insert/delete), which may also change the existing elements.

Each element in a list is composed of one or more fields (a smallest piece of information referenced in a programming language).

LISTS …

The operations on lists include -

- Combine two or more lists to form another list.
- Split a list into number of other lists.
- Copy a list.
- Determine number of elements in a list.
- Sort the elements of a list into ascending/descending order, depending on certain values of one or more fields within an element.
- Search a list for an element which contains a field having a certain value.



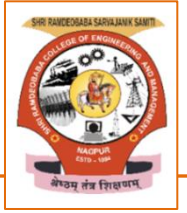
FILES

A file is typically a large list that is stored in the external memory of a computer (e.g., magnetic tape, magnetic disk).

A file may be used as a repository for list items (commonly called records) that are accessed infrequently and/or must be stored between various invocations of a program.

The access depends on file organizations (sequential / indexed / relative).

A file is the property of secondary storage.



STACK

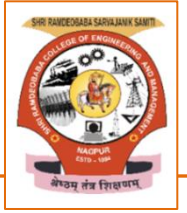
A stack is an ordered collection of elements, that allows additions and removals from one end only – the top.

A stack is referred as the last-in first out (LIFO) structure.

The operation of adding an element to a stack is called a push(), whereas that of removing an element from it is called a pop().

A stack is a list and can be checked for being empty using isEmpty() operation.

When a stack is implemented using arrays isFull() checks whether the top has exceeded the size bounds.



QUEUE

A queue is an ordered collection of elements, that allows additions from one end – the rear and removals from the other end – the front.

A queue is referred as the first-in first out (FIFO) structure.

The operation of adding an element to a queue is called a insert(), whereas that of removing an element from it is called a delete().

As like a stack, a queue is a list and can be checked for being empty using isEmpty().

When a queue is implemented using arrays isFull() checks whether all locations are occupied.

TREE

A tree is a hierarchical structure representing a parent-child relationship.

It is organized as collection of nodes. A special node – root, represents the top of the hierarchy and indicates the base address of the tree.

Each node of a tree can have zero or more children.

The intermediate nodes may have children and can have exactly one immediate parent.

The leaf nodes will never have children but will have exactly one immediate parent.

TREE …

In no circumstances, a node will have more than one immediate parents (this guarantees absence of cycles in the hierarchy).

Each sub-tree from a tree is always a tree.

GRAPH

A graph is a tree with cycles.

A graph G(V, E) is a set of vertices, V and a set of edges or arcs, E.

An edge connects a pair of vertices.

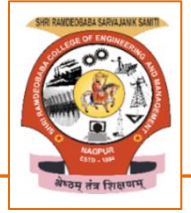
ALGORITHM

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.

An algorithm can be viewed as a tool for solving a well-specified computational problem. The statement of the problem specifies the desired input/output relationship.

The algorithm describes a specific computational procedure for achieving this input/output relationship.



The function SUM_OF_DIGITS which computes the sum of individual digits of a number, is given below –

```
FUNCTION SUM_OF_DIGITS (NUM)
1.  SUM ← 0
2.  While NUM <> 0
3.     SUM ← SUM + Call MOD(NUM, 10)
4.     NUM ← NUM / 10
5.  Return (SUM)
```

An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. It must satisfy following criteria -

1. Input : There are zero or more quantities that are externally supplied.
2. Output : At least one quantity is produced.
3. Definiteness : Each instruction is clear and unambiguous.
4. Finiteness : If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite number of steps.
5. Effectiveness : Every instruction must be basic enough to be carried out, in principle, by a person using only pencil and paper. It is not enough that each operation be definite as in (3), but also must be feasible.

Algorithms exist for many common problems.

It is very crucial for development of large-scale computer systems [Industrial Strength Software – ISS] to have an efficiently designed algorithm.

A program and an algorithm differs computationally on the ground that, the former does not have to satisfy condition [4].

For example,

An Operating System may continue in the wait loop until more jobs are entered.

It terminates when the system crashes or encounter a fatal error.

DATA TYPES AND ADTS

DATA TYPE

It is a collection of objects and set of operations that act on those objects.

ABSTRACT DATA TYPE [ADT]

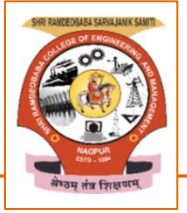
It is a data type that is organized in such a way that the specification of the objects and the specification of the operations on the objects is separated from the representation of the objects and the implementation of the operations.

ADA := through package

C++ := through class

C doesn't have an explicit mechanism to implement ADTs.

An ADT is implementation-independent.



The functions of a data type can be classified as -

- Creator/Constructor: These functions create a new instance of the designated type.
- Transformers: These functions also create an instance of the designated type, generally by using one or more other instances.
- Observers/Reporters: These functions provide information about an instance of the type, but they do not change the instance.

An ADT definition will include at least one function from each of these three categories.

THE ARRAY ADT

An array is a finite, ordered collection of data elements.

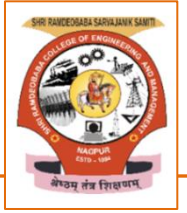
It is a set of index-value pairs, such that for index of an array, there exists an associated value.

OPERATIONS ON ARRAY

1. TRAVERSING

It involves accessing all elements in an array.

```
FUNCTION TRAVERSE_ARRAY (A)
Given an array, A of elements, this function accesses
every element of A.
1. For I ← 1 to A.Length
2.       Call Access (A[I])
```



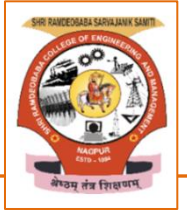
2. INSERTION

This operation involves inserting an element (i.e., a value) at a particular index of an array, subject to condition that an array is not full.

FUNCTION **INSERT_ELEMENT** (A, INDEX, VALUE)

Given an array, A of elements, this function inserts data value, VALUE at array index pointed by INDEX.

```
1. If A.Length = MX_LEN
2.     Print ('Error: Array bounds exceeded').
3.     Return
4. For I ← A.Length downto INDEX
5.     A[I+1] ← A[I]
6. A[INDEX] ← VALUE
```



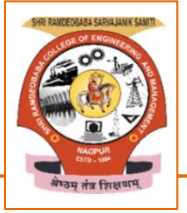
3. DELETION

This operation involves removing an element at a defined index from an array. It requires that an array is not empty.

FUNCTION **DELETE_ELEMENT** (A, INDEX)

Given an array, A of elements, this function removes a data value at array index pointed by INDEX.

```
1. If A.Length = 0
2.     Print ('Failure: Deletion from empty array').
3.     Return
4. For I ← INDEX to A.Length
5.     A[I] ← A[I+1]
6. A.Length ← A.Length - 1
```



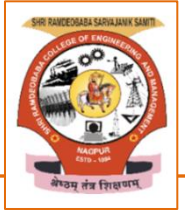
4. SEARCH

This operation allows locating an element in an array. It returns the index of the first occurrence of the element. If not found, it returns 0.

FUNCTION **SEARCH** (A, VALUE)

Given an array, A of elements, this function returns the array index at which the element, the VALUE exist. Otherwise, returns a zero.

```
1. For I ← 1 to A.Length
2.     If VALUE = A[I]
3.         Return I
4. Return 0
```



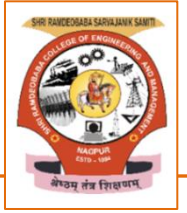
5. SORT

This operation allows ordering the elements of an array in ascending or descending sequence on the values of elements. The array is presumed to be non-empty.

FUNCTION **SORT** (A)

Given an array, A of elements, this function arranges array elements in an ascending sequence.

```
1. For I ← 1 to A.Length – 1
2.       For J ← I+1 to A.Length
3.           If A[I] > A[J]
4.               Call Swap(A[I], A[J])
```



5. LENGTH

This operation finds the length (i.e., number of elements) in an array. The array is presumed to be non-empty.

```
FUNCTION LENGTH (A)
```

Given an array, A of elements, this function computes its length. An index variable, I, is initialized to 1.

```
1. If A is Empty
2.       Return 0
3. While A[I] <> MN_VAL OR I <= MX_SIZE
4.       I ← I + 1
5. Return I
```

Thank You !!