VIVEKANANDA COLLEGE
THAKURPUKUR
KOLKATA-700063
NAAC ACCREDITED 'A' GRADE

Topic: Abstract Data Type (ADT)

Course Title: Data Structure

Paper: CMS-A-CC-2-3-TH

Unit: Introduction to data Structure

Semester: 2nd Semester

Name of the Teacher: Amitava Biswas

Name of the Department: Computer Science

Introduction and Types of ADT:

Objects (or data) may be stored on a computer using either:

- **Contiguous-based structures, or**
- **Node-based structures**

of which the array and the linked list are prototypical examples.

A data structure is a container which uses either contiguous- or node-based structures or both to store objects (in member variables or instance variables) and is associated with functions (member functions or methods) which allow manipulation of the stored objects.

A data structure may be directly implemented in any programming language; however, we will see that there are numerous different data structures which can store the same objects. Each data structure has advantages and disadvantages; for example, both arrays and singly linked lists may be used to store data in an order defined by the user. To demonstrate the differences:

Assuming we fill an array from the first position, an array allows the user to easily add or remove an object at the end of the array (assuming that all the entries have not yet been used),

A singly linked list allows the user to easily add or remove an object at the start of the list and a singly linked list with a pointer to the last node (the tail) allows the user to easily add an object at the end of the list.

There are significant differences between these two structures as well:

Arrays allow arbitrary access to the nth entry of the array, but

A linked list requires the program to step through n − 1 entries before accessing the nth entry.

Other differences include:

An **array** does not require new memory until it is full (in which case, usually all the entries must be copied over); but

A **singly linked list** requires new memory with each new node.

There are many other differences where an operation which can be done easily in linked lists requires significant effort with an array or vice versa.

Modifications may be made to each of these structures to reduce the complications required: an array size can be doubled, a singly linked list can be woven into an array to reduce the required number of memory allocations; however, there are some problems and restrictions which cannot be optimized away. An optimization in one area (increasing the speed or reduction in memory required by either one function or the data structure as a whole) may result in a detrimental effect elsewhere (a decrease in speed or increase in memory by another function or the data structure itself). Thus, rather than speaking about specific data structures, we need to step back and define models for specific data structures of interest to computer and software engineers.

An abstract data type or ADT (sometimes called an abstract data type) is a mathematical model of a data structure. It describes a container which holds a finite number of objects where the objects may be associated through a given binary relationship. The operations which may be performed on the container may be basic (e.g., insert, remove, etc.) or may be based on the relationship (e.g, given an object (possibly already in the container), find the next largest object). We will find that we cannot optimize all operations simultaneously and therefore we will have to give requirements for which operations must be optimal in both time and memory. Thus, to describe an abstract data structure we must:

Give the relationship between the objects being stored, and

Specify the operations which are to be optimized with respect to time and/or space.

Designing abstract data types”

Now we've seen how to implement an abstract data type. How do we choose when to use when, and what operations to give it? Let's try answering the second question first.

Parnas's Principle

Parnas's Principle is a statement of the fundamental idea of information hiding, which says that abstraction boundaries should be as narrow as possible:

The developer of a software component must provide the intended user with all the information needed to make effective use of the services provided by the component, and should provide no other information.

The developer of a software component must be provided with all the information necessary to carry out the given responsibilities assigned to the component, and should be provided with no other information.

For ADTs, this means we should provide as few functions for accessing

and modifying the ADT as we can get away with. The Sequence type we defined early has a particularly narrow interface; the developer of Sequence (whoever is writing sequence.c) needs to know nothing about what its user wants except for the arguments passed in to seq_create or seq_create_step, and the user only needs to be able to call seq_next. More complicated ADTs might provide larger sets of operations, but in general we know that an ADT provides a successful abstraction when the operations are all "natural" ones given our high-level description. If we find ourselves writing a lot of extra operations to let users tinker with the guts of our implementation, that may be a sign that either we aren't taking our abstraction barrier seriously enough, or that we need to put the abstraction barrier in a different place.

When to build an abstract data type?

The short answer: Whenever you can. A better answer: The best heuristic I know for deciding what ADTs to include in a program is to write down a description of how your program is going to work. For each noun or noun phrase in the description, either identify a built-in data type to implement it or design an abstract data type.

For example: a grade database maintains a list of students, and for each student it keeps a list of grades. So here we might want data types to represent:

- A list of students,
- A student,
- A list of grades,
- A grade.

If grades are simple, we might be able to make them just be ints (or maybe doubles); to be on the safe side, we should probably create a Grade type with a typedef. The other types are likely to be more complicated. Each student might have in addition to his or her grades a long list of other attributes, such as a name, an email address, etc. By wrapping students up as abstract data types we can extend these attributes if we need to, or allow for very general implementations (say, by allowing a student to have an arbitrary list of keyword-attribute pairs). The two kinds of lists are likely to be examples of sequence types; we'll be seeing a lot of ways to implement these as the course progresses. If we want to perform the same kinds of operations on both lists, we might want to try to implement them as a single list data type, which then is specialized to hold either students or grades; this is not always easy to do in C, but we'll see examples of how to do this, too.

Whether or not this set of four types is the set we will finally use, writing it down gives us a place to start writing our program. We can start writing interface files for each of the data types, and then evolve their implementations and the main program in parallel, adjusting the interfaces as we find that we have provided too little (or too much) data for each component to do what it must.