e-PGPathshala

Subject : Computer Science

Paper: Data Structures

Module: Introduction to Abstract Data Types

Module No: CS/DS/1

Quadrant 1 – e-text

Welcome to the e-PG Pathshala Lecture Series on Data Structures. This time we are going to talk about data structures from a different perspective

Learning Objectives

The learning objectives of the introductory module are as follows:

- To appreciate Programming perspectives
- To understand Programming Design principles
- To know about the different Data Types
- To understand Abstract data types

1.1 Introduction

Though most of you would have studied Data Structures, in this course we will be looking at Data Structure from a new perspective – the perspective of Abstract Data Type. Before we understand the concept of abstract data types let us relook at Programming which is essentially problem solving using a computer system. However though computers today are used to solve very complex problems – programming is not magic, the maxim remains that you cannot make a computer do something if you do not know how to do it yourself. Before you start programming, you need to understand the problem. Some of the important tips are, find out as much as you can about the problem by talking to the problem presenter, as far as possible reuse what has been done before and design the program expecting future reuse. The first step in understanding the problem is breaking the complex problems into sub-problems and then trying to determine what previous attempts at solving this problem or similar problems have succeeded and what attempts have failed and why they have failed. In order to appreciate the concept of Abstract Data Types we must first consider some important design principles.

1.2Design Principles

There are some principles that are to be followed when designing a good program. These include abstraction, encapsulation, modularity and hierarchy. Let us consider each of the principles one by one.

1.2.1 Abstraction

Here are two general definitions of Abstraction.

“Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon those similarities and to ignore for the time being the differences.” Tony Hoare

“An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of objects and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.” Grady Booch

In other words abstraction is also described as the purposeful suppression, or hiding of some details of a process or an artifact, in order to bring out more clearly other aspects, details, or structure. In the context of programming the use of Abstract Data Types (which we will discuss later) or the use of objects encourages abstraction.The principle of Abstractioncan be described as the process of determining the relevant and essential properties and features of an entity or what we call an object while ignoring nonessential details with respect to the way the entity is implemented and also focuses on obtaining interfaces (outside view) of objects.Moreover relevant properties are defined by how the entity is to be used and manipulated. For example, an auto salesperson views a car from the standpoint of its selling features, while a mechanic views the car from the standpoint of the systems that require maintenance.

1.2.2 Encapsulation

Here are two definitions of Encapsulation.

“Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.” Grady Booch

“Encapsulation is a mechanism used to hide the data, internal structure, and implementation details of an object. All interaction with the object is through a public interface of operations.” Craig Larman

Encapsulation is a mechanism by which we restrict the access to some of the object's components, as well as binding the data and methods operating on the data. The principle of **Encapsulation** can be described *as the separation of objects based on external and internal aspects.* In other words only the external aspects of an object need to be visible to other objects in the system, while the internal aspects are details that should not affect other parts of the system and need not be visible to them. Encapsulation is a technique for packaging the information in such a way as to hide what should be hidden, and make visible what is intended to be visible that ishiding implementation details.Hiding the internal aspects of an object essentially means that they can be changed without requiring changes to other system parts.

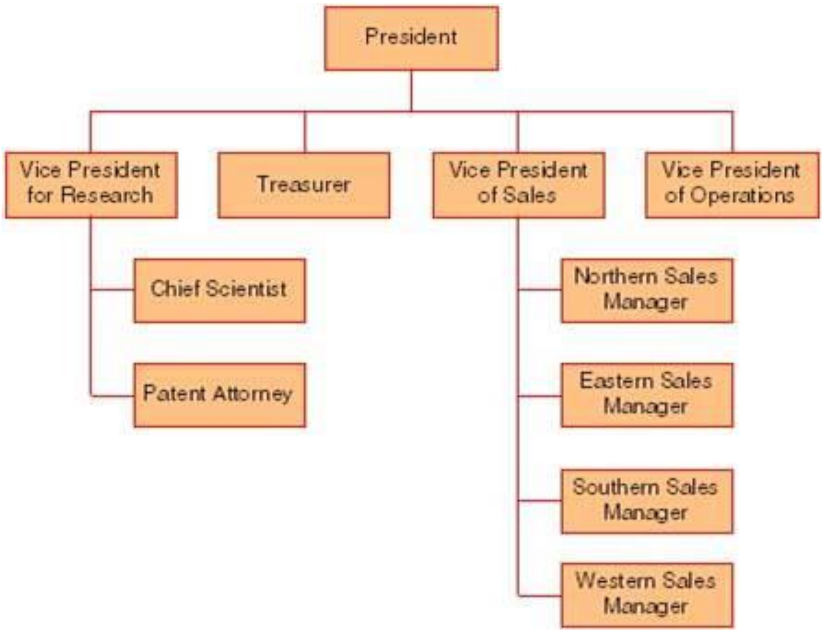

Figure 1.2 Management hierarchy

1.2.3 Hierarchy

The principle of**Hierarchy** can be described as an useful way of ordering abstractions from the most general to least general based on some relationship between them. Hierarchies can help us understand complex objects. Hierarchy makes all the abstractions easier to understand because it exposes the relationship of the characteristics and behaviors they have in common. An example of a management hierarchy is shown in Figure 1.2 which is based on who reports to who. A specific type of hierarchical ordering based on natural relationships is called taxonomy. An example of taxonomy is that of musical instruments that organizes instruments by how they produce their sound.Before we define Abstract Data Types let us recall the description of traditional data structures.

1.3Data Structures

Let us recall the description of data structures. Data structures are a systematic way of organizing and manipulating data. In other words a*data structure* structures data, usually more than one piece of data and should provide legal operations on the data.There are aspects of data organization and functions for manipulating data. Data structures are conceptual and concrete ways to organize data for *efficient storage*and *manipulation.* Data structures are basically used in the implementation of *efficient algorithms.*

1.3.1 Important Data Structures

Before we begin discussing the details of Abstract Data types or ADT as they are called let us list some common linear data structures some of which we will study later on from the perspective of Abstract Data Types. These include fixed-size arrays, variable size linked-lists, stacks where we add to top and remove from top, queues where we add to back (rear) and remove from front and priority queue where we add anywhere but remove the element with highest priority. Next we go on to discuss the concept of a data type before go on to discuss the details of abstract data types.

1.4 What is a data type?

Data types are associated with a set of entities or objects and a set of operations. Each object is called an instance of the data type. Some of these objects are sufficiently important to be provided with a special name.Operations can be realized via operators, functions, procedures, methods, and special syntax (depending on the implementing language). Of course each object must have some representation (not necessarily known to the user of the data type) and each operation must have some implementation (also not necessarily known to the user of the data type). Data types are basically of two types, opaque data typesand transparent data types.

Opaque data types are data types in which the representation is not known to the user. Representation can be changed without affecting the user since these data types encapsulate the operations. Without encapsulation you cannot have opaque data types. This forces the program designer to consider the operations more carefully and allows less restrictive designs which are easier to extend and modify. Design is generally carried out with the expectation that the data type will be placed in a library of types available for everyone to use.

Transparent data types are data types in which the representation is profitably known to the user, in other words the encoding is directly accessible and/or modifiable by the user. This allows the user to manipulate the data types because the representation is known to the user. In the next section we go on to discuss abstract data types.

1.5Abstract Data Types -Concept of Abstraction

Let us first recall the notion of abstraction. An *abstraction* is a view or representation of a data type that includes only the most significant attributes. The concept of abstraction from this viewpoint means you know what a data type can do but the details of how it is done is hidden.

The concept of *abstraction* is fundamental to programming. Programming languages generally have supported *process abstraction*in the form of subprograms and some of the more recent programming languages do also support *dataabstraction.*In simple terms with data abstraction***what*** you can do with the data is separated from ***how***it is represented. Data abstraction, or abstract data types, is a programming methodology where one defines not only the data structure to be used, but the processes to manipulate that data structure.

An Abstract Data Type is one that is defined in terms of the operations that it supports (i.e., that can be performed upon it) rather than in terms of its structure or implementation. An *abstract data type* is a user-defined data type that satisfies the following two conditions. The first condition is that the representation and operations of the type are defined in a single syntactic unit. The second condition is that the representation of the type is hidden from the program units that use these objects so that the only operations possible are those provided in the data type's definition.

1.5.1 Advantages of Data Abstraction

Now let us look at some advantages of data abstraction. The advantage of the first conditionof data abstraction (knowing what a data can do) is better program organization, modifiability(everything associated with a data structure is together), and separate compilation. The advantage of the second condition (representation is hidden) is reliability because by hiding the data representations, user code does notdepend on the representation thus allowing the representation to be changed without affecting user code.

1.6Abstract Data Type- ADT

Some data is associated with a set of operations and mathematical abstractions have been an integral part of programming languages, abstractions which we have been using without giving it much thought like integer,Boolean, etc. The basic idea ADTs is a logical view of the data objects together with specifications of the operations required to create and manipulate them. Designing and coding an ADT once enables the reuse of the ADT. Any changes in the implementation are transparent to the other modules. Just like an algorithm is described using pseudo-code, a data structure can be described using ADT. The meaning of 'abstract' in Latin means to 'pull out' and in our context abstract essentially means to pull out the essentials and to defer or hide the details. In other words abstraction emphasizes essentials and defers the details, making engineering artifacts easier to use. Figure 1.3 shows the difference between primitive data types and the abstract data types.

In other words ADT can be described as follows :
1. Declaration of data
2. Declaration of operations
3. Encapsulation of data and operations

Summing up ADT is a type for encapsulating related data and is abstract in the sense that it hides distracting implementation details. Now what are the programming languages that naturally support ADTs? Objects are a perfect programming mechanism to create ADTs.Given below are some examples of ADT:

***Set* ADT**
- A set of elements
- Operations: *union, intersection, size* and *complement*

Q*ueue* ADT
- A set of sequences of elements
- Operations: c*reate empty queue, insert, examine, delete*, and d*estroy queue*

A very important point to be noted with respect to ADTs is that *two ADTs are different if they have the same underlying model but different operations*. Thus for example a different set ADT can be defined with only the *union* and *find* operations. It is to be noted that the appropriateness of an implementation depends very much on the operations to be performed.

1.6.1 Need for ADT

As we have already discussed, ADT allows implementation to be changed without violating ADT definitions. ADTs provide us a standard list of structures to use and study. They give us building blocks and tools which can conveniently be used by programmers. ADT is the foundation for the design of Object Oriented Programming languages like C++, Java etc. They adopt concepts of abstraction, encapsulation, modularity, hierarchy which comes with their own advantages.

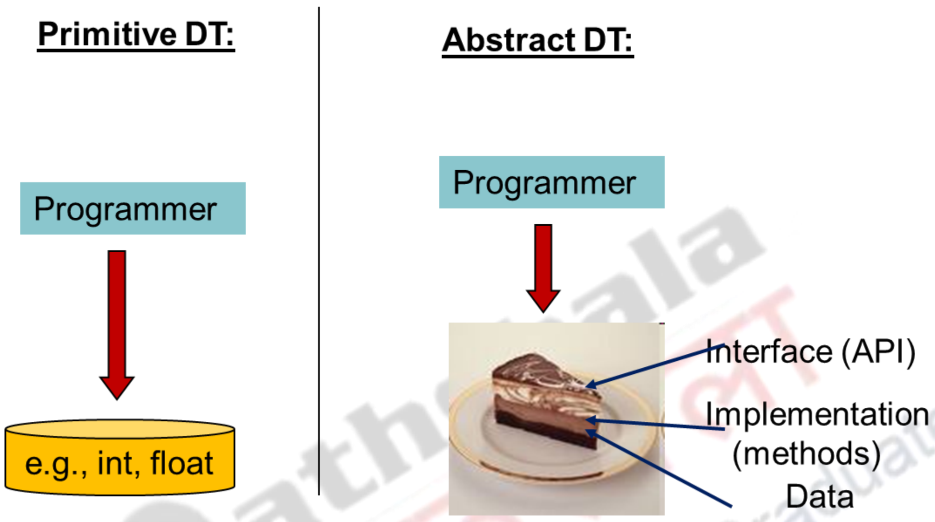

Figure 1.3 Primitive Data Type vs. Abstract Data Types

1.6.2 Pros and Consof ADT

Now let us list some of the advantages of the use of ADT. One of the important advantages is that implementation of the ADT is separate from its use. By design ADT is modular and we can design one module for one ADT. ADTs are easier to debug, and also makes it easier for several people to work simultaneously. The Code written for an ADT can be reused in different applications and it allows the concept of information hiding. Logical unit can be designed to do a specific job and implementation details can be changed without affecting user programs. All the above advantages allowsrapid prototying since the prototype can be designed with simple ADT implementations, then tuned later when necessary. One of the disadvantages of ADT based design is the loss of efficiency since the representation and implementation details are hidden and the knowledge of these details cannot be exploited to improve efficiency.

1.7Basic Concepts of ADT

Let us revisit some of the concepts of abstraction , but now look at them from the viewpoint of ADT.

1.7.1Modularity

The first concept we will discuss is the concept of modularity. Modularity keeps the complexity of a large program manageable by systematically controlling the interaction of its components. It helps to isolates errors and eliminates redundancies. A modular program is essentially easier to write, easier to read and easier to modify.

1.7.2Information Hiding

The next concept we discuss is information hiding. ADT hides the implementation details of the operations and the data from the users of the ADT. In other words makes these details inaccessible from outside the module. Users can use the operations of an ADT without knowing how the operation is actually implemented. Examples include a deck of playing cards and a set of index cards containing contact information and telephone numbers stored in your cellular phone.

1.7.3Data Abstraction

The most important concept of an ADT is data abstraction. It asks you to think ***what* you can do** to a collection of data independently of ***how* you do it.** It allows you to develop each data structure in relative isolation from the rest of the solution and is essentially a natural extension of procedural abstraction.

1.7.4Encapsulation

In the context of ADT encapsulation allows operation on an ADT only by calling the appropriate function. There will be no mention of *how* the set of operations is implemented. In essence the definition of the type and all operations on that type can be localized to one section of the program. After this point we can treat the ADT as a primitive type since we are no longer concerned with the underlying implementation.

1.8Design of Abstract Data Type

Thefirst step in the design process is the specification of the problem where the design of an ADT should evolve naturally during the process. The questions to be asked while designing the ADT are the type of data the problem requires and the associated operations required. For complex abstract data types, the behavior of the operations can be specified using axioms. An example of an axiom is given in Figure 1.4.

Axiom: A mathematical rule

Ex. : (aList.createList()).size() = 0

Figure 1.4 Example of an Axiom

The second step in the design is the development of a real or imaginary application to test the specification.Missing or incomplete operations are found as a side-effect of trying to use the specification in the application.

The third and final step is the implementation of the ADT where we decide on a suitable representation, implement the operations and finally test, debug, and revise the code.

1.9 Typical Operations on Abstract Data Types

Before we complete this module we will look at some typical operations that will define ADTs irrespective of the type of ADTs. These include creating a data collection, adding data to a data collection, removing data from a data collection and asking questions about the data in a data collection.

Create operation
It is always necessary to create an object before it can be used. For example, in Java this is done using the class constructors.

Copy operation
The availability of this operation depends on the particular ADT. In many cases it is not needed or desired. If present, the meaning (semantics) of the operation also depends on the particular ADT. In some cases copy means make a true copy of the object and all its data fields, and all their data fields, and so on, and in other cases it may mean to simply make a new reference to an object.In other words, the reference to the object is being copied, not the object itself. In this case there is only one object and it is shared among all the references to it. This makes sense for objects that occupy large amounts of memory and in many other cases as well. Both types of operation can even be included in the same ADT. In some languages the copy operation can have explicit and implicit versions. For example in Java the implicit operation, defined by assignment or method argument passing, always copies references but it is possible to make other kinds of explicit copies using a copy constructor or by overriding the clone method inherited from the Object class.

Destroy operation
Since objects take up space in memory it is necessary to reclaim this space when an object is no longer needed. This operation is often called the *destroy* operation. In Java there is no explicit destroy operation since the built-in garbage collector takes on this responsibility: when there are no more references to an object it is eventually garbage-collected.

Modification operations
Every object of an ADT encapsulates data and for some ADTs we need operations that can modify this data. These operations act on objects and change one or more of their data fields. Sometimes they are called **mutator**operations. If an ADT has no mutator operations then the state cannot be changed after an object has been created and the ADT is said to be **immutable**, otherwise it is **mutable**.

Inquiry operations
An inquiry operation inspects or retrieves the value of a data field without modification. It is possible to completely hide all or part of the internal state of an object simply by not providing the corresponding inquiry operations.

1.10 Pre- conditions and post-conditions

When working with ADTs, pre-conditions and post-conditions effectively clarifies and documents your thinking process. In addition, the operation definition, along with the pre and post-conditions, supply all the information required by others to utilise the interface of the data type. In other words, specifying what the operation does along with the pre and post-conditions enables **procedural abstraction,** an important concept of ADT.

Pre-condition statement or set of statements outlines a condition that should be true, or conditions that should be true, when the operation is called. The operation is not guaranteed to perform the way it should unless the preconditions have been met. On the other hand, post-condition statement or statements describe the condition that will be true when the operation completes its task. If the operation is correct and the pre-condition(s) had been met, then the post-condition is guaranteed to be true. In addition, an operation is also associated with invariants that specify properties of the ADT that are not changed by this operation. Let us assume that the operation is specified as a function. Pre-conditions usually focus on the inputs to a function while post-conditions usually focus on the output of a function and any side effect.

This type of specification where pre-conditions and post-conditions are specified explicitly is known as 'programming by contract'. The pre and post-conditions define what the operation expects before it is executed and what it promises to have done when it finishes. No interface specification is complete without pre and post-conditions. What you are doing is forming a contract between the ADT and its user. The pre-conditions define a state of the which the client guarantees will be true before calling any operation, whereas the post-conditions define the state that will be guaranteed to be true when it completes. Pre and post-conditions are like a statement in law; they need to be precisely stated, and are of no use if ambiguous.

When we take an example of an operation then we can define it as function that takes a parameter x and returns the square root of x. The precondition stated here is that x must be greater than or equal to zero for this operation to give the expected output while post-condition specifies what is expected to be true after the operation is completed. Given below (Figure 1.4) is an example that illustrates this concept.

Example

```
Void write_sqrt( double x)
//   Precondition:  x  >=  0.
//   Postcondition:  The square root of x has
//   been written to the standard output.
...
Which of these function calls meet the precondition?
write_sqrt( -10 );
write_sqrt( 0 );
write_sqrt( 5.6 );
Here the first function alone does not satisfy the precondition.
```

Figure 1.4 Example of Precondition

It is the responsibility of the person who uses the operation to ensure that the precondition is valid when the function is called and actually counts on the precondition being valid. Similarly the post condition must be valid after the function is completed.

Later on when these ADTs are actually implemented using a specific representation careful programmers ensure that when a function is written they make every effort to detect when a precondition has been violated. When such a violation is detected, the programmer makes sure that an error message is printed and the program is halted, rather than causing a disaster due to precondition violation.

1.10.1 Advantages of Using Preconditions and Post conditions

There are some advantages when such pre-conditions and post-conditions are explicitly specified. These statements succinctly describe the behaviour of a function without cluttering up the thinking with details of how the function works. This allows programmers to later re-implement the function in a new way, however programs that depend only on the precondition / postcondition will still work with no changes.

1.11 Language Requirements for ADTs

As is the case with process abstraction, ADTs can be supported directly by programming languages. Some of the requirements that a programming language should posses in order to program using ADTs include the availability of a syntactic unit in which to encapsulate the type definition, a method of making type names and subprogram headers visible to the clients, while hiding actual definitions and some primitive operations that must be built into the language processor. Objects are one way to implement ADTs.

Summary

- Discussed important programming design principles
- Understood the different data types
- Explained Abstract Data types and their usage
- Listed the various Data Structures