

## Discussion of Data

The notion of data is very general. Recently it is being defined in terms of computers:

**data.....**

1. known facts or things used as a basis for inference or reckoning.

2. quantities or characters operated on by a computer etc ..

(*Concise Oxford Dictionary 9<sup>th</sup> ed. 1995*)

Distinctions between *data*, *information* and *knowledge* are still fluid - this can be a source of confusion.

*Traditionally....*

Programming = manipulating data stored in computer memories using computer algorithms

(e.g. 'Algorithms + Data structures = Programs', Wirth)

*Currently .....*

Programming is concerned with building computer models of parts of the world (systems which include humans and devices of all kinds). Interaction and communication are essential in such systems. Data needs to be flexible (e.g. sensitive to context and viewpoint) and dynamic (e.g. change in real time, distributed over networks, integrity and security).

A **data type** is

- (i) a domain of allowed values,
- (ii) a representation for the domain of values
- (iii) a set of operations on those values.

Simple types are those in which the values do not have parts (atomic). Java primitive types such as boolean, byte, char, short, int, long, float, double, are simple.

A **structured data type** is one in which

- (i) the values can be decomposed into component elements
- (ii) there is a set of relationships (structure) between the component elements.

These are also known as *container* types. In Java the reference types array, Vector, Stack, etc are structured types.

An **abstract data type (ADT)** is a data type (simple or structured) which is viewed independently of its representation in a language or machine.

A **specification** of a data type is a statement of the essence of the type - independent of its implementation. It must include the domain of values, any structure they have, and the set of operations. It should *not* include commitment to any representation.

## Levels of data abstraction :

Application-oriented abstractions

Algorithms over data structures

High-level languages

Assembly language

Machine code

Note the level of abstraction is **orthogonal** to the kind of data type (simple or structured). For example:

	<i>Abstract</i>	<i>Virtual</i>	<i>Physical</i>
<i>Structured</i>	student list	float []	array on SUN
<i>Simple</i>	class size	int	SUN integer

## Examples of Structure and of Specification

### Array

Linear array : finite set of similar elements referenced by indices (such as 0,1,2,3...)

2-d array : finite set of similar elements referenced by pairs of indices, etc.

### tuple/record

ordered sequence of elements, not necessarily all of the same type.

### Linked list

An ordered sequence of nodes, each of which is an ordered pair consisting of a data value and an address for the next node.

### Tree (binary)

Finite set  $T$  of nodes such that either  $T$  is empty or  $T$  contains a distinguished node  $t$  called the *root* and  $T - t$  is an ordered pair of disjoint binary trees  $(L,R)$ .

If  $T$  has a root then  $L$  and  $R$  are respectively the left and right subtrees of  $T$ .

## **Stack** (“LIFO” list)

A last-in first-out linear list of data items. Items are added (*push*) and removed (*pop*) at the same end (the *top* of the stack).

## **Queue** (“FIFO” list)

A first-in first-out linear list of data items. Items are added and removed at opposite ends (*tail* and *head* of the queue respectively).

## **Graph**

A pair of sets  $G = (V, E)$  where

- (i)  $V$  is a set of *vertices* (nodes or points)
- (ii)  $E$  is a set of unordered pairs of nodes in a  $V$  (i.e. a subset of  $V \times V$ ).  $E$  is called the *edges* of  $G$ .

## **Typical data structure operations:**

- (i) *traversing* - accessing (visiting) each record exactly once to process items
- (ii) *searching* - locating the record with a given key value, or finding locations whose data values satisfy some condition.
- (iii) *adding* - introducing a new record to a structure.

- (iv) *deleting* - removing a record from a structure.
- (v) *sorting* - arranging records in some logical order.
- (vi) *merging* - combining the records in two sorted files into a single sorted file.

## Specifying a simple ADT:

### ADT Colour

**Domain:** set of possible values = { R, Y, B, G, O, V }

### Operations:

mix (c1,c2 : colour) : colour     *{combine c1 and c2}*

**pre:** c1 and c2 are primary colours

**post:** mix(c1,c2) is the colour formed by mixing c1 and c2 in equal amounts.

primary (c:colour) : boolean     *{is c a primary colour?}*

**pre:** none

**post:** primary(c) is true iff c is a primary colour

form (c:colour; c1,c2: colour)

*{what colours make up c?}*

**pre:** c is not a primary colour

**post:** c1 and c2 are the colours that make up c

assign (c1: colour; c2: colour)

*{assign value c2 to c1}*

**pre:** none

**post:** c1 has the value of c2

### Points to note:

**pre** is an abbreviation for **pre-condition**.

A pre-condition is an assertion that must be true in order for the operation to execute correctly. We do not specify what happens if mix is applied to non-primary colours.

Implicit pre-conditions – we assume c1 and c2 are values of type colour.

**post** is an abbreviation for **post-condition**.

A post-condition is an assertion that is guaranteed to be true after the operation has been completed. It describes the effects (combination of computing a value and - perhaps - affecting the state of variables supplied as arguments (a “side effect”)) of performing an operation. Not **how** the effect is achieved, but **what** effect is achieved by the action.

Example of boolean function of 2 variables.

Example of letterstring.

## Some advantages of ADTs:

- **Simplicity**

Abstraction separates essential qualities of data, structure and operations from inessential details of representation and implementation

- **Integrity**

Limiting the class of operations that can be performed can guarantee that integrity constraints are met. Users cannot perform operations that destroy data integrity.

- **Implementation independence**

We can change the implementation without requiring the user to change their programs; as long as declared operations of the ADT still meet their specification

Implementation via ADTs uses the

### **Encapsulation (information hiding) principle**

*... the only way in which a user is allowed to access or manipulate the data is through the specified operations*

The ADT approach separates the specification from the implementation.