Abstract Data Types

An **abstract data type (ADT)** is a mathematical model of a data type. It formally specifies what it *means* for data entities (things represented in computer memory) to be of a certain *type*, but it does not specify an *implementation* (a description of how such a type should be represented in code or in hardware). One can think of an ADT as a *goal*; one can think of an implementation of an ADT (using **data structures**) as a *solution* that achieves that goal.

An ADT T is defined by:

1. a set of **values**, the class (or *extension*) of all of the possible abstract entities of type T (not their representations in computer memory but the abstract entities themselves, e.g. the integers, the alphabetic characters, the days of the week, every conceivable list, every conceivable tree, etc.);
2. a set of **operations** (an interface of functions) that are necessarily defined over and thus can be performed on entities of type T; and
3. a set of **constraints**, which the operations must satisfy and which, in turn, define the functional behavior of said operations.

An ADT can also, more simply, be thought of as a set of **objects**, where an object is a value paired with a set of operations defined over that value. Objects of a type T would span a range of values but would each have the same set of operations subject to the same set of constraints. Note that a complicated object could be defined in terms of simple objects and could thus have a value that is *composed of* or that somehow *contains* other values (e.g. a list of names has a value and the names within the list also have values).

All ADTs have a create() operation (or *constructor*) that yields an **instance** of the ADT that is distinct from any other instances of the ADT that might currently be in use (including those with the same value, as each instance must have a unique memory address and name). The abstract relation between an ADT and an instance is known as *instantiation.* Each instance (represented concretely in computer memory) would have an abstract counterpart in the class of objects described by an ADT and would behave accordingly.

We will now examine several ADTs of varying complexity. We will also discuss which data structures are commonly used in implementing each and why.

Variables

An abstract variable V is a symbolic name to which a value of some type (or, in mathematical terms, an element of some set), may be assigned. It is a mutable entity (i.e. its value can be changed). All mutable ADTs (those defined in an imperative style) rely on the basic concept of a mutable variable.

Values:

- Any set of conceptual entities that can be specified by a type (e.g. the set of rational numbers, the set of five-letter words playable in Scrabble, the set of potato varieties, etc.).

Operations:

- store(V, x), where x is a value of unspecified type; and
- fetch(V), which yields a value.

The Turing machine model provides the theoretical basis for storing and fetching values to and from variables.

Constraints:

- fetch(V) always returns the value x used in the most recent store(V, x) operation on the same variable V. Fetching the value of a variable before any value is stored under it is generally considered an invalid operation.
- If U and V are distinct variables, the sequence {store(U, x); store(V, y)} is equivalent to {store(V, y}; store(U, x)}.

Implementation:

The main challenge in implementing an ADT is figuring out how to give sequences of bits *meaning*. This is done by selecting a storage location in computer memory (identified by a **memory address**), assigning a **symbolic name** to that location, and associating the bits stored at that location with a **type**. The type would determine the size of the data (the number of bits to read beyond the original storage location) and the value of the data (the abstract object that the bits are meant to represent). Operations are implemented by programming **subroutines** such that the constraints are satisfied. A single ADT may be implemented in many different ways. ADTs serve to unify the many possible implementations under a common functional description.

Simple ADTs

An abstract variable V may optionally restrict values x to members of a set described by some *type*. Simple types describe conceptually atomic entities, things that cannot be broken down into meaningful parts. Objects of these types represent the simplest pieces of information.

Simple ADTs are implemented in programming languages as primitive types (or, simply, **primitives**). These types are built into programming languages at the hardware level (i.e. the choice of implementation is abstracted from the programmer for the sake of speed and convenience).

The Numeric Types: Integral, Rational, Real, and Complex Numbers

Values:

- Integral numbers
  - Natural numbers (unsigned integers): 0, 1, 2, ...
  - Integers (signed integers): ..., -2, -1, 0, 1, 2, ...
- Rational numbers: ...
- Real numbers: ...
- Complex numbers: ...

One could argue that rational numbers and complex numbers have parts (numerator/denominator and real/imaginary), but one could also argue instead that they are conceptually atomic. 1/2 and 2/4 represent the same rational number; the choice of numerator and denominator is arbitrary as long as the ratio is preserved. Similarly, a complex number is as atomic as a real number, if real numbers are defined as complex numbers with an imaginary part equal to zero. In the end, whether these numbers are simple or not depends on one's conceptual ontology, and they are implemented as primitives (or not) at the discretion of the programming language designer.

Operations:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Equivalence (==)
- Less than (<)
- Greater than (>)
- Less than or equal (<=)
- Greater than or equal (>=)
- Increment (++) (only for integers)
- Decrement (–) (only for integers)
- Unary negation (-) (only for signed integers)

Division by zero should throw an error because the result is undefined. (There is a **precondition** for the division operation that constrains the second operand to all integers other than zero.)

Implementation:

Numeric data types can either be **exact-value** or **approximate-value**. Integrals and rationals can be represented exactly. (Rational data types typically implement rational numbers as ordered pairs of integers and implement algebraic operations with this format in mind). Because the vast majority of real numbers cannot be represented in finite computer memory, real data types can only represent approximations of real numbers. Typically, fixed-point or floating-point types are used to implement approximations of the real numbers.

Unsigned integers are represented in the **binary numeral system** and may be **little or big endian**. Signed integers are typically represented in **two's complement binary**, although sign magnitude and one's complement are also possible. Floats are typically represented according to **IEEE 754**. These are practical choices made for ease of computation by digital electronics. They are abstracted from the programmer, who is concerned only with values, not with how values are implemented.

Because computer memory is finite, only a finite range of exact values can be represented. For this reason, many exact-value data types of varying size may represent a single ADT. Integral data types include: nibble, byte, short, int, long, long long, quad, etc. Likewise, approximate values are limited to a certain precision. Real data types include: float, double, long double, etc.

The Logical Type: Boolean Values

Values:

- Boolean: true, false.

Operations:

- Conjunction (AND)
- Disjunction (OR)
- Exclusive disjunction (XOR)
- Equivalence (==)
- Negation (NOT)

Implementation:

Because information in computers is typically **word-aligned**, boolean values are often represented similarly to 32-bit integers. Zero would map to false, and all other values would map to true.

The Symbolic Type: Character

Values:

- Symbols in a character set (e.g. letters in an alphabet, ASCII, UTF-8)

Operations:

- upper(c)
- lower(c)

Implementation:

Characters are usually implemented as integers (such as in The C Programming Language), so they may also have arithmetic operations.

Reference

Values:

- Handle: any kind of key that allows access to a value
- Pointer: memory addresses (keys to values/objects stored in memory)
- Index: relative positions (keys to values/objects stored in an ordered structure whose address in memory is known)

Operations:

- Address-of (&). This is the create() function for pointers. It takes an entity of type T stored in memory and returns its address (which is of type T*, a pointer to a value of type T)..
- Dereference (*)

Composite ADTs

Composition is an abstract relation that allows for composite types, which describe

Containers

A container is a grouping of a variable number of objects (generally of the same type).