Abstract Data Types

Step 1

As programmers, we are often given a task in non-technical terms, and it is up to us to figure out how to actually implement the higher-level idea. How do we choose what tools to use to perform this task? Typically, the job at hand has certain requirements, so we scour the documentation of our favorite language (or alternatively just search on Google and pick the first StackOverflow result, who are we kidding) and pick something that has the functions that satisfy our job's requirements.

For example, say we want to store the grades of numerous students in a class in a way that we can query the data structure with a student's name and have it return the student's grade. Clearly, some type of a "map" data structure, which maps "keys" (student names) to "values" (grades), would do the trick. If we're not too worried about performance, we don't necessarily care about *how* the data structure executes these tasks: we just care that it gets the job done.

What we are describing, a model for data types where the data type is defined by its *behavior* from the point of view of a *user* of the data (i.e., by what functions the user claims it needs to have) is an **Abstract Data Type (ADT)**.

Step 2

We just introduced a term, "abstract data type," but this course is about "data structures," so what's the difference? The two terms sound so similar, but they actually have very different implications. As I mentioned, an **Abstract Data Type** is defined by what functions it should be able to perform, but it does not at all depend on how it actually goes about doing those functions (i.e., it is not implementation-specific). A **Data Structure**, on the other hand, is a concrete representation of data: it consists of all of the nuts and bolts behind how the data are represented (how they are stored in memory, algorithms for performing various operations, etc.).

For example, say a user wants us to design a "set" container to contain integers. The user decide that this container should have the following functions:

- An "insert" function that will return True upon insertion of a new element, or False if the element already exists
- A "find" function that will return True if an element exists, otherwise False
- A "union" function that will add all of the elements from another "set" into this "set"

The user has successfully described what functionality the "set" container will have, without any implementation-specific details (i.e., *how* those three functions described will work). Consequently, this "set" container is an **Abstract Data Type**. It is now up to us, the programmers, to figure out a way to *actually* implement this **Abstract Data Type.**

So can we figure out the time complexities of any of the three functions described in the "set" container above? Unfortunately, no! Because an **Abstract Data Type** *only* describes functionality, not implementation, the time complexities of these functions completely depends on how the programmer chooses to *implement* the "set" (i.e., what **Data Structure** we choose to use to back this container).

Step 3

EXERCISE BREAK: Which of the following statements are true about an **Abstract Data Type**? (Select all that apply)

Step 4

In short, an **Abstract Data Type** simply *describes* a set of features, and based on the features we wish to have, we need to choose an appropriate **Data Structure** to use as the backbone to implement the ADT.

For example, what if a user wanted an **ADT** to store a list of songs? When we listen to music, do we always necessarily want to iterate through the songs in the order in which they appear? Typically, we like having the ability to choose a specific song, which could appear somewhere in the middle of the list of songs. As a result, it might make sense for us, the programmers, to use an array-based structure to implement this **ADT** because having random access could prove useful. However, we very well could use a Linked List to implement this **ADT** instead! The functionality would still be correct, but it might not be as fast as using an array-based structure.

In the next sections, we will discuss some fundamental **Abstract Data Types** and will discuss different approaches we can use to implement them (as well as the trade-offs of each approach).