I’m happy you’ve stumbled across my data-structures tutorial! If you are not proponent of fancy elitist Computer Science hoopla, or if you’re not sure why people make such a big deal about this domain of software development please read on! If you already drink the Kool-aide on this subject feel free to proceed to the following heading.

So, Why are data structures important?

Many people will tell you data-structures are the fundamental subject that makes a computer scientist who they are, and the problem-solving skills learned through taking such a course are imperative in the journey of honing one’s algorithmic skills. Though I agree with this I’d prefer give some more practical reasons for attaining fluency with data structures.

1. Use the right tool for the right problem:

You wouldn’t hammer a screw right? So why would you not learn to use the proper structure for a problem? Array’s are awesome structures that provide a host of reason’s to use them, but if you have a problem that requires many insertions and deletions, you’d be better off using a linked list. Furthermore learning about datastructures, you’ll probably come across some structures or algorithms you didn’t previously know of, and will add one more tool to your toolbox of computing.

1. Effiicient Problem solving:

Let’s say you’re solving the canonical sorting problem. Let’s say you’re using the selection sort, bubble sort, or insertion sort. These algorithms are going to produce quadratic runtime (e.g. O(N^2)), while if you only knew what a heap was you could achieve a much better near logarithmic runtime (e.g. O(n log(n))). The point being levering data-structures often produces much more efficient algorithms when used properly.

1. The job market and the dreaded technical interview:

Most software companies will give you technical interview questions, and the vast majority of companies draw from data-structures and algorithmic problems when probing candidates for their technical capacities. This is a tried and true method for screening candidates because: it illustrates how candidates approach a problem, they are language agnostic, and they don’t require you to understand the company’s software stack.

1. Knowing and understanding data-structures provides you the capacity to write your own:

when the out of the box implementation doesn’t suit the problem

Let’s say our company has chosen the programing language, “Crazy Academic Functional Lambda Language”, to implement an optimized concurrent application for your company’s server architecture. Perhaps this language doesn’t have a suitable hash-table implementation for your problem because it rehashes too often and is running slow. Well good thing you know data-structures because you can implement your own hash-table that caters to your needs. I admit this last point is the least likely of the 4 to occur, but it is certainly within the realm of possibility.

---------------------------------------------------------------------------------------------------------------------------

Before you get started on your way to becoming an efficient developer, there are a few concepts that you may or may not have been introduced to that are integral to your understanding of the concepts that I will introduce you to in the coming tutorials.

1. Abstract Data Types (ADT’s)

Abstract Data Types were invented by one of our awesome women in computing (not to mention one of the first!) Barbara Liskov. An Abstract datatype is:

“”” a [mathematical model](https://en.wikipedia.org/wiki/Mathematical_model) for [data types](https://en.wikipedia.org/wiki/Data_type), where a data type is defined by its behavior ([semantics](https://en.wikipedia.org/wiki/Semantics_(computer_science))) from the point of view of a *user* of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

“””

<https://en.wikipedia.org/wiki/Abstract_data_type>

So let’s unpack this a little: data is defined by behavior, values, and operations independent of implementation. Think of a simple list. There are certain operations you will always want to perform on lists irrespective of whether the list contains Strings, numbers, objects, or another list. You most likely want to be capable of adding and removing items from the list, retrieving items, checking if the list contains an item, or checking the size of the list. You know how the list should interact with data irrespective of the way the structure is implemented in code, and irrespective of the data itself. A list is a list am I right?

This is why we have ADT’s. This is why we developers can refer to a List, Stack, Queue, Map, Graph, Tree, etc. and another developer will know what we are talking about. A Haskell programmer, a C programmer, and a python programmer can all have a conversation about these structures without understanding the nuances of the specific implementations in one language or another because they all have the same operations, and they treat data the same way, irrespective of the data type. In my tutorials the data structures I will explaining will be implemented starting with a java interface, an example of which you can see below where I’m introducing parameterized classes.

1. What is Parameterized programming, or a parameterized class

Parameterized programming is a technique that is used to make your code and classes more portable and reusable. When parameterizing our classes we provide generic datatypes to our methods that allow us to implement our data-structures and algorithms without knowing which datatypes they will operate on. By doing so we don’t have to write an algorithm for every type: Integer, Float, String, Object, etc. We use generics and then the programmer using the data structure implements the methods for the class upon which their algorithm is operating. For instance for lists you need to know if an object T is equal to another object, so the programmer would implement the method T equals(T x) where T is the object that fits the desired use case of your class or algorithm. Note if you are using java classes the methods have already been implemented for you, but these methods must be implemented for a custom class that will be passed to a data structure.

To combine these two paradigms let me introduce you to the parameterized List interface:

interface List<T> {  
 void add(T x);  
 T remove(int i);  
 boolean remove(T x);  
 T get(int i);  
 boolean contains(T x);  
 int size();  
 default boolean isEmpty() {  
 return size() == 0;  
 }  
}

Here we use the angular brackets to parameterize the list interface so that it may be instantiated with many different types, provided they are not primitive data types (e.g. not int float etc.). When you want to use primitive data types you must use the wrapper classes of these types (e.g. Integer, Float, Boolean).

1. What is Big-O notation and time and space complexity

Big-O notation is a tool used to compare the efficiency of various algorithms. Big-O notation is always written with respect to N because it is representing the time or complexity with respect to the given input size. This notation may be used to represent both the time complexity and the space complexity of an algorithm. Time complexity is the time it takes the algorithm to execute with respect to the input size, while Space complexity is the space that must be allocated to execute the algorithm with respect to the input size. Though we can talk about best case, average case, and worst case run time. For the purposes of my data structures tutorials, we will most often use Big-O to denote the worst case complexity of an algorithm. (Algorithm nerds please don’t disown me for this!)

The following table and chart denote the most common big-O complexities.

* Table
* Chart

Don’t forget to mention we drop constants & drop low order terms when dominated by high order terms

3 \* O(1) -> O(1)

X = 2 \* 5 / 5;

Y = 5- 7 + 2;

Sout(X + Y)

O(log(N))

* Binary Search

O(1) + O(N) -> O(N)

Y = 5 \* 10

For(int I = 0; i < N; i++ )

Sout(i\*y);

O(N \* log(N))

* Merge sort

O(N) + O(N^2) -> O(N^2)

For(int i = 0; i < N; i++)

sout(i)

for(int i = 0; i < N; i++)

for(int j = 0; j < N; j++)

sout(I \* j)