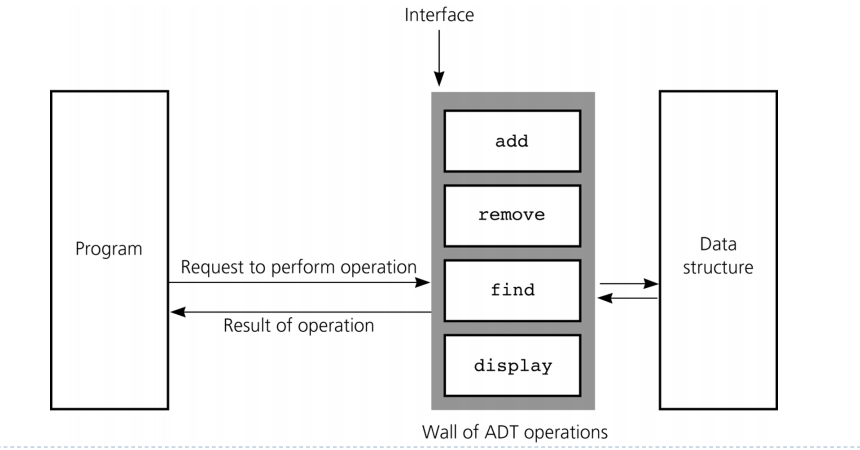
Abstract data type

Abstract Data type is a class of entities with a set of values and a set of operations describing their behaviour. Abstract types of data are mathematical models of a set of data values or information that share similar behaviour or attributes and can be defined and described irrespective of actual implementation. And Usually, abstract forms of data are used in algorithms. The abstract data type is specified instead of being implemented in terms of its data items or its related operations. And Abstract data types, abstract data structures or forms are specified indirectly only by the operations that can be performed on them and by mathematical constraints on their results.

ADT are the data types which are made up of or composed of these primitive data types as these ADT can be application or implementation specific, they could be created on need basis.



Here are some examples

* **stack:** operations are "push an item onto the stack", "pop an item from the stack", "ask if the stack is empty"; implementation may be as array, linked list,
* **queue:** operations are "add to the end of the queue", "delete from the beginning of the queue", "ask if the queue is empty"; implementation may be as array or linked list

ADT is to an interface, what a data structure is to a class.

A few examples:

**ADT** – List

**DS**: Array List, Linked List, Vector List

**ADT** – Map

**DS:** Hash Map, Tree Map, Sorted Map

**ADT** – Set

**DS** –Hash Set, Tree Set, Linked Hash Set

The fundamental difference between the abstract data type (ADT) and the concrete data type is that the latter helps us to look at the actual representation, while the latter hides from us the representation. An ADT may be pure ADT or Updatable ADT. One where all operations are pure functions is a pure ADT. It means that there are no side effects of operations. In addition, input arguments are not changed or revised there. We simply use these arguments to produce output, which are ADT (or other forms of fresh values). Many forms of concrete are simple. For example, no integers operation actually modifies an integer. Alternatively, fresh production is provided by all operations like' +.'

Data Structure

A data structure is a specific way of organizing information in a system for efficient use. as an example, using the array data structure, we can store a list of items with the same data type.

|  |  |  |  |
| --- | --- | --- | --- |
| U | B | F | D |

0 1 2 3

I have listed below popular data structures

* array
* Stack
* Queue
* Binary Tree
* Graph
* Heap
* Metrics
* Linked List

Queue Data Structure

A Queue is a linear structure that follows a specific order performing the operations. First in First Out (FIFO) is ordered. A good example of a queue is any customer queue for a service where the first-coming user is first served. The difference is in reducing stacks and queues. In a stack we delete the newly added element; in a queue we remove the newest added object.

Sorting algorithms

A Sorting Algorithm is used to rearrange the elements of a given sequence or list based on a comparison operator. The comparison operator is used in the respective data structure to determine the new order of element.

**Measuring and comparing bubble sort and quick sort algorithm**

A general way of measuring an algorithm is using Big O, which takes a simplified mathematical approach.

In order to explain this at a very basic level, a simple linear search of an integer list has a linear (n) worst case. Big O. Eg: for(int i = 0; i < sizeofarray; ++i) if(array[i] == to\_find) return i;

At the worst case this would take i iterations often number is referred to as n in big o - so we call it n or linear complexity algorithm.

Something like a bubblesort algorithm is a loop within a loop so we have n \* n complexity = n^2 or quadratic complexity.

If we just consider sorting, comparing like with like, quicksort is more effective than quadratic complexity. It is the complexity of n log n so you can find quicksort to be ' better ' than bubblesort.

So when evaluating your algorithm think about it in terms of n. Is there a loop? how many? The fewer the better. No loops even better - constant big o

Task 2

**Advantage of encapsulation and information hiding when using an ADT**

Data encapsulation, also referred to as data hiding is the mechanism by which the client is kept hidden from implementing class info. Only by performing special functions commonly called methods can the user perform a restricted collection of operations on the secret members of the class. System behaviours are decided by the system developer, who must be careful not to make the methods either too open or too restrictive. This idea to hide the user's details and provide a restricted

Using data encapsulation has the advantage when the class structure adjusts, but the functionality maintains the same. For example, the developer may choose to implement it with an array that is hidden from the class user to construct a stack class that may contain integers. The programmer then writes the methods push() and pop() bringing integers into the array and removing them respectively from the list. Such techniques will be made available to the consumer. If the client tries to reach the array directly, it will result in a compile time error.

Now, should the programmer decide to change the implementation of the stack to a linked list, simply replace the array with a linked list and rewrite the push() and pop () methods to control the linked list instead of the array.

**There is a difference between "abstract data type" and "abstract class"**

An abstract class is one that may not have definitions for all the methods it defines. You therefore cannot directly instantiate an abstract class. You have to create a subclass and then instantiate that.

An abstract data type is a model of a certain kind of data structure e.g. a Stack. A Stack has push () and pop () operations and that have well-defined behaviour.

The abstract data type (ADT) itself refers to this model, not any particular implementation in any particular programming language or paradigm. You could implement a Stack in an object-oriented language, but you could also implement it in a functional programming language.

ADTs allow discussion about the properties of Stacks, Queues etc that hold for all correct implementations of the ADT.

Task 4

**Asymptotic analysis of an algorithm**

We test an algorithm's performance in terms of input size in Asymptotic Analysis we don't calculate the actual running time. We measure how an algorithm's time or space increases with the size of the input.

For example, let's look at the search problem in a sorted array, searching for a given item. Linear Search is one way to search, and Binary Search is another way. To understand how Asymptotic Analysis solves the above problems in testing algorithms, let's assume we're running Linear Search on a fast computer and Binary Search on a slow computer. The fast machine can take less time for small values of the input array size n. Nevertheless, after some input array size length, the Binary Search would undoubtedly begin to take less time compared to the Linear Search, even though the Binary Search is running on a slow computer.

The reason is Binary Search's growth order with respect to logarithmic input size while Linear Search's growth order is linear. After certain values of input size therefore, the machine-dependent constants can always be ignored.

**Efficiency of algorithm**

An algorithm can solve a problem we need, then the algorithm is a great algorithm. Nevertheless, an algorithm's efficiency is calculated in two areas: **time and space**. An algorithm may have good time efficiency, but low space efficiency or an algorithm may have good space efficiency but bad time efficiency.

**Time efficiency** is the time the algorithm requires to solve a problem, and the **size / space efficiency** is the size of the memory (allocated to variables) that the algorithm needs to solve the problem. Therefore, the efficiency of the algorithm is calculated in time efficiency and performance of space / size.

So, for that, we typically evaluate the algorithm's worst case for time and/or space. Best Case and Normal scenarios are other cases. We're doing worst case analysis because it gives the algorithm's lowest possible efficiency and we're pretty sure the algorithm won't go beyond that curve either for time or size / space.

## **Advantages of Abstract Data Types**

Abstract data type offers several advantages over concreate data type

* **Representation independence**

Most of the program is independent of the representation of the abstract data type so that it is possible to improve representation without destroying the entire program.

* **Modularity**

Through independent representation, the different parts of a program become less dependent on other components and how those other components are implemented

* **Interchangeability of parts**

Similar abstract data type implementations may have different performance features. For abstract data types, using an application of its data types that will be more effective for that particular part of the program becomes simpler for each part of a program.

Example:

Java's standard libraries supply several different implementations of its Map data type. The TreeMap implementation might be more efficient when a total ordering on the keys can be computed quickly but a good hash value is hard to compute efficiently. The HashMap implementation might be more efficient when hash values can be computed quickly and there is no obvious ordering on keys. The part of a program that creates a Map can decide which implementation to use. The parts of a program that deal with a created Map don't have to know how it was implemented; once created, it's just a Map.

If it weren't for abstract data types, every part of the program that uses a Map would have to be written twice, with one version to deal with TreeMap implementations and another version to deal with HashMap implementations.