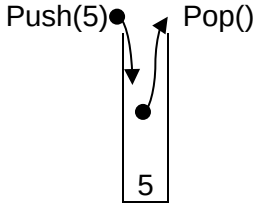
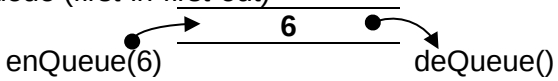
1. Introduction to Data Structure and Algorithm

Primitive Data Types:
Primitive Data Types are basic data types of a programming language.
These refer to two things: a data item with certain characteristics and the permissible operations on that data.
For example,
The operations + (addition), - (subtraction), * (multiplication), / (division) can be applied to the primitive data types integer and float.
The operations && (and), || (or), ! (not) can be applied to the primitive data type Boolean.

Abstract Data Types (ADT):
ADT is a specification of a mathematical set of data and the set of operations that can be performed on the data.
The set of operations that define an ADT is referred to as its interface.
ADT focuses on what it does and ignores how it does its job. That is, ADT is independent of any particular implementation.
E.g.:
1. stack (last-in-first-out)



2. queue (first-in-first-out)



Data Structures:
Data structure is the arrangement of data in a computer memory/storage.
Data structure is the implementation of ADT such as stack, queue, linked list, tree, and graph.
When working with certain data structures we need to know how to insert new data, how to search for a specific item, and how to delete a specific item.
Data structure helps for efficient programming.
Data structure reduces complexity of the program and its calculations.
There are two types of data structures: Linear data structure and nonlinear data structure

Linear Data Structure	***Nonlinear Data Structure***
Data Organized Sequentially (one after the other)	Data organized non sequentially
Easy to implement because the computer memory is also organized as linear fashion	Difficult to implement
E.g.: Array, Stack, Queue	E.g.: Tree, Graph

What is Algorithm?
An algorithm is a step by step procedure for solving a problem in a finite amount of time. Many algorithms apply directly to a specific data structures.
Properties of an algorithm are given below:
- Should be written in simple English
- Should be unambiguous, precise and clear.
- Should provide the correct solutions
- Should have an end point
- Should have finite number of steps

Efficiency of an Algorithm:
Some algorithms are more efficient than others. We would prefer to choose an efficient algorithm.
Running time of algorithms typically depends on the input set, and its size (n).

Big 'O' Notation:

We can say that a function is "of the order of n", which can be written as O(n) to describe the upper bound on the number of operations. This is called Big-Oh notation.

Some common orders are:

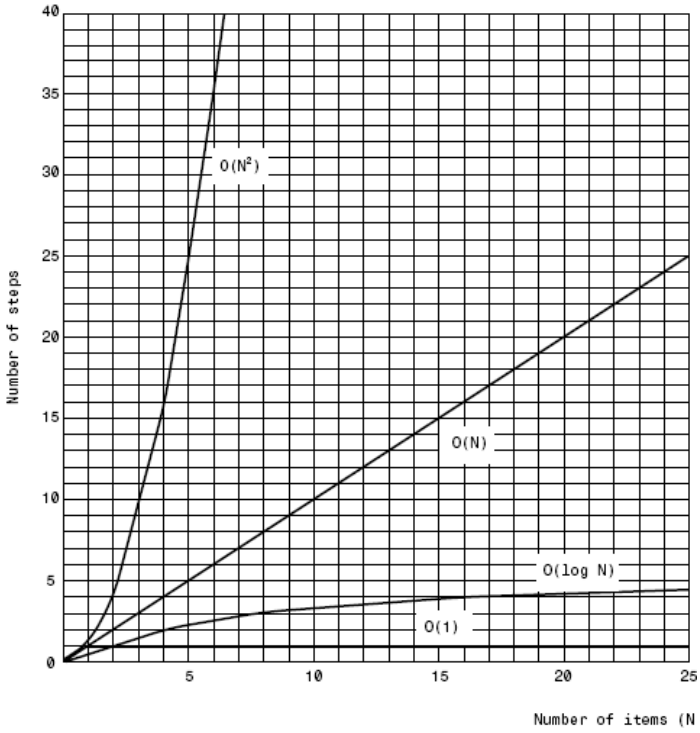
- O(1) constant (the size of n has no effect)
- O(log n) logarithmic
- O(n log n)
- O(n)
- O(n^2) quadratic
- O(n^3) cubic
- O(2^n) exponential

If the number of operations is 500, then the big 'O' notation is O(1).

If the number of operations is 9n + 200, then the big 'O' notation is O(n).

If the number of operations is $n^3 + 3n + 8$, then the big 'O' notation is O(n^3).

Generally we can take O(1) is faster than O(n) and O(n) is faster than O(n^3).



Best, Worst and Average Cases:

Best case efficiency is the minimum number of steps that an algorithm can take any collection of data values.

Worst case efficiency is the maximum number of steps that an algorithm can take for any collection of data values.

Average case efficiency

- is the efficiency averaged on all possible inputs
- must assume a distribution of the input
- is normally assumed for uniform distribution (all keys are equally probable)

For example, when we search for an element in a list sequentially,

- In best case, the cost is 1 compare. That is O(1).
- In worst case, the cost is n compares. That is O(n).
- In the average case, the cost may be (n+1)/2. That is O(n).

Example of Analysis:

Consider the following algorithm.

Input: array X of n integers
Output: array A of prefix averages

$A \leftarrow$ new array of n integers
for $i \leftarrow 0$ to $n-1$ **do**
  $sum \leftarrow X[0]$
  for $j \leftarrow 1$ to i **do**
    $sum \leftarrow sum + X[j]$
  end for
  $A[i] \leftarrow s/(i+1)$
end for
return array A

Find statement executed most of the time. In this case statement inside inner loop.

Total number of time inner loop is executed n(n-1)/2 times.

Therefore, running time is O(n^2).