University of Portsmouth
BSc (Hons) Computer Science
Second Year

Data Structures and Algorithms (`DSALG`)

M21270
September 2023 - January 2024
20 Credits

Thomas Boxall
`up2108121@myport.ac.uk`

Contents

Page 1

Async lecture - Introduction to Data Structures and ADT

2023-09-30

1.1 Data Structures

A *Data Structure* is a way to store and organise data in order to facilitate access and modification. There is no single data structure which is perfect for every application - we need to choose the best for whatever we are creating.

There are two parts to a data structure: a collection of elements, each of which is either a data type of another data structure; and a set of associations or relationships (the structure) involving the collection of elements.

1.1.1 Classification of Data Structures

Data structures can be classified based on their predecessor and successor.

Name	**Predecessor**	**Successor**	**Examples**
Linear	unique	unique	stack, queue
Hierarchical	unique	many	family tree, management structure
Graph	many	many	railway map, social network
Set Structure	no	no	DSALG class

Table 1.1: Classifications of Data Structures

1.1.2 Choosing the right Data Structure

When choosing a data structure, it is important to analyse the problem, determine the basic operations needed and select the most efficient data structure. Choosing the right data structure will make the operations simple & efficient and choosing the wrong data structure will make your operations cumbersome and inefficient.

1.1.3 CRUD

CRUD Operations: Create, Read, Update and Delete are the basic operations which all data structures must be able to do. It is common for a data structure to use a different name to refer to the operation, however.

1.2 Abstract Data Type

An Abstract Data Type (ADT) is a collection of data and associated methods stored as a single module. The data within an ADT cannot be accessed directly, it must be accessed indirectly through its methods. An ADT consists of: the data structure itself; methods to access the data structure; methods to modify the data structure; and internal methods (which are not accessible from outside the ADT).

1.3 Algorithms

An *Algorithm* is any well-defined computational procedure that takes some data, or set of data as input and produces some data or set of data as output. It is the sequence of computational steps which are gone through that transforms the input into the output which is the algorithm. The algorithm must process the data efficiently (both in terms of time and space).

1.3.1 Classifications of Algorithms

There are a number of different classifications of algorithms - four are shown below. Definitions are from *National Institute of Standards and Technology - Dictionary of Algorithms and Data Structures*

Brute-Force Algorithm

An algorithm that inefficiently solves a problem, often by trying every one of a wide range of possible solutions. E.g. exhaustive search.

Divide & Conquer Algorithm

An algorithm which solves a problem either directly because that instance is easy (typically, this would be because the instance is small) or by dividing it into two or more smaller instances. Each of these smaller instances is recursively solved, and the solutions are combined to produce a solution for the original instance.

Backtracking Algorithms

An algorithm that finds a solution by trying one of several choices. If the choice proves incorrect, computation backtracks or restarts at the point of choice and tries another choice. It is often convenient to maintain choice points and alternate choices using recursion.

Greedy Algorithms

An algorithm that always takes the best immediate, or local, solution while finding an overall answer. Greedy algorithms find the overall, or globally, optimal solution for some optimisation problems, but may find less-than optimal solutions for some instances of other problems.

1.4 Stack Abstract Data Type

A *stack* is a collection of objects where only the most recently inserted object (`top`) can be removed at any time. A stack is linear and operates in Last In First Out (LIFO).

Stacks must support the following operations.

`push` add an item to the top of the stack

`pop` remove an item from the top of the stack

`peek` examine item at the top of the stack

`empty` determine if the stack is empty

`full` determine if the stack is full

An application using a stack will expect the ADT to thrown an exception if: a push operation is requested on a full stack; or a push / pop operation is requested on an empty stack. The stack manages its own storage, therefore an application which uses a stack is not concerned with how the storage used by the stack is managed. In general, an ADT is not interested in the application using the ADT.

Stacks can be implemented using a static array and have a number of uses, including: matching brackets in arithmetic expressions; recursive algorithms; and evaluating arithmetic expressions.

1.5 Queue Abstract Data Type

A Queue is a collection of objects organised such that the first object to be stored in the queue is the first to be removed and so on. It is a linear data structure with elements - inserted at one end (the tail) and removed from the other (the head). A queue operates in First In First Out (FIFO).

Queues must support the following operations:

`enqueue` add item to the queue's tail

`dequeue` remove item from the queue's head

`full` check if queue is full (therefore total number of elements exceeds max capacity)

`empty` check if queue is empty

`first` check the first element (the head) in the queue

1.5.1 Implementations of a Queue

There are three different implementations of a queue, all of which can use a static array.

Fixed Head

The head of the queue is fixed, this means it will always be in index 0 of the static array. When an element is dequeued, the rest of the elements in the queue must be shuffled along so that the new head is in index 0. This is extremely time inefficient for large queues however quite space efficient as there won't be "dead space" at one end of the queue.

Mobile Head

The head of the queue is mobile, this means it can be in any index of the array. When an element is dequeued, the rest of the elements in the queue stay where they are and the head pointer is updated to represent the new head's index. This is more time efficient however not space efficient as you may end up with a lot of "dead space" where the head of the queue used to be.

Circular Queue

The queue is circular in a logical, not physical, way. This means the head and tail can be anywhere in the array. If the head is not at the start of the array and the space is needed, the tail will loop around to use the space at the start of the static array.

Page 2

Async lecture - Tools of the Trade I: Efficiency & BigO

2023-09-30

BigO Notation is used to define the efficiency of an algorithm quantitatively. This is a very helpful tool to have when designing algorithms as it allows us to compare multiple algorithms to and understand which is the best one to use.

Name	**Notation**	**Description**
Constant	$O(1)$	Algorithm always executes in the same amount of time regardless of the size of the dataset.
Logarithmic	$O(\log_n)$	Algorithm which halves the dataset with each pass, efficient with large datasets, increases execution time at a slower rate than that at which the dataset size increases.
Linear	$O(n)$	Algorithm whose performance declines as the data set grows, reduces efficiency with increasingly large dataset.
Loglinear	$O(N \log_n)$	Algorithm that divides a dataset but can be solved using concurrency on independent divided lists.
Polynomial	$O(N^2)$	Algorithm whose performance is proportional to the size of the dataset, efficiency significantly reduces with increasingly large datasets.
Exponential	$O(2^n)$	Algorithm that doubles with each addition to the dataset in each pass, very inefficient.

Table 2.1: BigO Notation complexity values, listed best to worse

BigO doesn't look at the exact number of operations, it looks at when the size of a problem approximates infinity therefore two very similar algorithms which are slightly different may have the same Big O despite one having double the number of operations.

2.1 Calculating The BigO Value

1. Determine the basic operations (including: assignment, multiplication, addition, subtraction, division, etc)
2. Count how many basic operations there are in the algorithm (some basic algebraic addition required here!)
3. Convert the total number of operations to BigO (done by: ignoring the less dominant terms; and ignoring the constant coefficient - i.e. $2n + 1$ becomes n).