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).

Page 3

Async lecture - Iterative Algorithms and Efficiency

2023-10-05

3.1 Searching Algorithms

When searching a dataset to find the item you are looking for, the aim of an efficient searching algorithm is to exclude elements to reduce the searching space after each comparison.

3.1.1 Sorted vs Unsorted Data

Sorted data, when the data is in either ascending or descending order, makes it easier to locate the item you are searching for. Depending on the algorithm, when using sorted data, we can exclude more elements from the list to search.

Unsorted data, when the data is in a random order, makes it harder to locate the item you are searching for. This is because you have to examine (in the worst case) every element in the array you are searching to realise that the item you are searching for is not in the array.

3.1.2 Sequential (Linear) Search

1. Start at the beginning
2. Check every element of the array in turn until item located or the end of the array reached (therefore item not located)

This search excludes one data item at a time, which is not great. It works on sorted and unsorted data.

The best case BigO is $O(1)$. This is the case where the first item we examine is the item we want to find.

The worst case BigO is $O(n)$ when the searched value is the last element in the array or not in the array, n comparisons are required.

The average case BigO is $O(\frac{n}{2})$ because if the data is distributed randomly, each element has an equal chance to be the one searched for.

Overall, the BigO for a sequential search is $O(n)$.

3.1.3 Binary Search

1. Check the middle element of the array

2. If not found, work out which half of the array the item could be located in and exclude the half which it won't be included in
3. Repeat by searching the half array which may contain the required item by examining the middle element and eliminating half the array
4. Process repeated on the halved array until either find the item or determine that it doesn't exist.

A binary search only works on a sorted array.

The best case BigO is $O(1)$. This is achieved when the value we are searching for is the middle element, hence it is found on the first comparison.

The worst case BigO is $O(\log_2 n)$, which is achieved when the value we are searching for is not found in the array. We therefore need $1 + \log_2 n$ comparisons which gets converted into BigO.

The average case is $O(\log_2 n)$, if the data is distributed randomly - each element has an equal chance to be the one searched for.

The binary search algorithm has an additional cost - it requires a sorted sequence of data items, which incurs cost if the data is not already sorted.

3.2 Sorting Algorithms

A sorting algorithm aims to make comparisons between data items and swap them according to the desired order of the items. All sorting algorithms will involve two basic steps:

1. Compare items
2. Swap elements

3.2.1 Selection Sort

A selection sort works by sorting the array one item at a time. It divides the list into two parts. The sorted section (on the left) is initially empty and the unsorted section (on the right) is initially the complete unsorted list. The smallest (or largest, depending on if we want ascending or descending) is selected from the unsorted array and swapped with the left-most element from the unsorted section. The item is now in the correct final position within the ascending / descending order. The sorted part has increased in size by 1. The process is iterated on the unsorted part of the array until all items have been considered, the array will now be sorted.

The best case BigO is $O(n^2)$, when the array is already sorted hence each element only has to be compared to its direct neighbours to establish this.

The worst case is $O(n^2)$.

Selection sorts are unsuitable for large data sets.

3.2.2 Bubble Sort

A bubble sort works by repeatedly passing through the list, swapping adjacent items if they are in the wrong order. If sorting into ascending (or descending) order, the largest (or smallest) item will be bubbled to the end of the array, hence the name. The process is iterated on the whole list until all items have been considered, therefore the whole list will now be sorted.

Without using a flag, the best case BigO is $O(n^2)$. The worst case BigO is also $O(n^2)$.

When a flag (variable which denotes when the array is sorted, enabling sorting to stop as soon as array detected as sorted) is used, the best case BigO is $O(n)$ and the worst case BigO is $O(n^2)$.

With or without the flag, the algorithm is very slow for large datasets.

3.2.3 Insertion Sort

An insertion sort works by sorting the list one item at a time. THe list is split into a sorted and non-sorted part. With each iteration, the next element waiting to be sorted (in the unsorted part) is inserted in its correct location within the sorted list. The process iteration on whole list until all items have been considered; when completed, the list will now be sorted.

The best case BigO is $O(n)$, when the data is already sorted. The worst case BigO is $O(n^2)$.

This algorithm is efficient for sorting nearly-sorted lists.

Page 4

Async lecture - Tools of the Trade II: Recursion

📅 2023-10-05 🕘 🎓

4.1 Introduction

Computers are better at repeating themselves than humans are. Humans get bored, computers don't.

Iteration is explicit repetition of code. We often use for and while loops to repeat sections of code, both of which use control variables to control the repetitions.

Recursion is an alternative method of repeating code, where the code is repeated implicitly. Recursion occurs when a function or method calls itself.

4.2 Recursion

Recursion is a technique whereby a problem is expressed as sub-problems in a similar / same form to the original problem but smaller in scope. The sub-problems only differ in input or size. Shown below is an example of a recursive algorithm, that never ends.

```
recursive_print_example(i){
    print(i)
    recursive_print_example(i+1)
}
```

Recursion is applied to problems where: a solution is easy to specify for certain conditions (the stopping case, which is required or the program will continue indefinitely); and rules for proceeding to a new state which is either a stopping case or eventually leads to a stopping case (recursive steps) are identified.

Shown below is an actual recursive algorithm with its output:

```
cheers(int times){
    print("hip)
    if (times > 0){
        cheers(times - 1)
    }
    print("hooray")
}
//outputs: hip hip hip hooray hooray hooray
```

4.3 Pitfalls of Recursion

Recursion must always be used with care and understanding. It is possible to write compact and elegant recursive programs that fail spectacularly at runtime. The main pitfalls of recursive algorithms are as follows:

- Forgetting the stopping case
- Failure to reach a stopping case
- Excessive use of space
- Excessive repeated computations

There are some times, when you really shouldn't use recursions:

- When the algorithm / data structure is not naturally suited to recursion
- When the recursive solution is not shorter and understandable than the linear solution
- When the recursive solution doesn't run in acceptable time limits and / or space limits
- When the intermediate states of the algorithm don't pass the same data to / from them.

Recursion to be continued next week.