Hugh Baldwin
up2157117

Data Structures and Algorithms

M21270
TB1

University of Portsmouth
BSc Computer Science
2nd Year

Contents

- Mid-Unit assessment (30%) on 8th November
- Exam (70%) in main assessment period

Lecture - Workshop 1 and 2 Async

16:00 27/09/23 Dalin Zhou

Data Structures

- A data structure is a way to store and organise data
  - A collection of elements
  - A set of relations between the elements
- Classification of data structures
  - Linear
    - Unique predecessor and successor
    - e.g. Stack, Queue, etc
  - Hierarchical
    - Unique predecessor, many successors
    - e.g. Family tree, management structre, etc
  - Graph
    - Many predecessors, many successors
    - e.g. Railway or road map, social network
  - Set
    - No predecessor or successor
    - e.g. A class of students
- Static vs Dynamic
  - A static data structure has a fixed size, which cannot be exceeded and must be allocated in it's entirety when created
  - A dynamic data structure has a dynamic size, which can change at runtime and only uses as much memory as is needed for it's contents
- Most data structures follow CRUD(S) for their basic operations;
  - Create - Add a new element
  - Read - Read an existing element
  - Update - Modify an existing element
  - Destroy (or Delete) - Remove an existing element
  - (Search) - Search for an element matching certain search conditions
- Each program would need a different data structre - there is no one-size-fits-all

Abstract Data Types

- An ADT is a collection of data and associated methods
- The data in the ADT cannot be directly accessed, only by using the methods defined in the ADT
- Stacks
  - A stack is a collection of objects in which only the top-most element can be modified at any time
  - This means it is a LIFO (Last-in first-out) structure
  - A stack must implement the following methods:
    * Push - Add an item to the top of the stack
    * Pop - Remove the item from the top of the stack
    * Peek - Examine the item at the top of the stack
    * Empty - Determine if the stack is empty
    * Full - Determine if the stack is full
    * A stack is typically implemented using an array, which is hidden behind the methods of the ADT
- Queues
  - A queue is a collection of objects in which the object which has been in the queue for the longest time is removed first
  - This means it is a FIFO (First-in first-out) structure
  - A queue must implement the following methods:
    * Enqueue - Add a new item to the tail of the queue
    * Dequeue - Remove the item at the head of the queue
  - A queue is typically implemented using an array, which is hidden behind the methods of the ADT
  - There are two methods of implementing a queue
    * Fixed head
      · The head of the queue is always at the 0th position in the array, and elements are shifted as they are dequeued
    * Dynamic head
      · The head of the queue changes to the index of the current head of the queue, and elements are not shifted
      · Since this leaves empty spaces before the head, space may be wasted. To solve this, a circular array could be used

Algorithms

- An algorithm is a procedure that takes a value or set there of as input and produces a value or set there of as an output
- A sequence of computational steps that transforms the input into the output
- Ideally, we would always use the most efficient algorithm that is available
- Classification of algorithms
  - Brute-force
  - Divide and conquer
  - Backtracking
  - Greedy
  - etc

Big-O Notation

- To determine the Big-O of an algorithm
  - Count how many basic operations (assignment, addition, multiplication and division) there are for the worst-case scenario (e.g. 10 items in a stack of length 10)
  - Ignore the less dominant terms of this equation
  - Ignore any constant coefficients
  - The remaining terms become the Big-O complexity of the algorithm
- Example 1:
  - There are $2n + 1$ operations for an algorithm
  - We can ignore the $+1$ as this has less impact, leaving $2n$
  - We can ignore the constant coefficient 2, leaving n
  - So the Big-O notation would be $O(n)$
- Example 2:
  - There are $2n^2 + n + 1$ operations for an algorithm
  - We can ignore the $+1$, leaving $2n^2 + n$
  - We can ignore the $+n$, leaving $2n^2$
  - We can ignore the 2, leaving n^2
  - So the Big-O notation would be $O(n^2)$
- Order of dominance:
  - Constant < Linear < Logarithmic < Quadratic < Cubic < ...
  - $O(1) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) <$ ...

Lecture - Workshop 3 and 4 Async

15:00 05/10/23 Dalin Zhou

Searching Algorithms

- It is much easier to search for an item if they are already sorted
- Linear Search
  - In a linear search, you start at index 0 and keep looking through the items in the array until the item is found
  - Since the worst-case would be checking every item in the list, the worst-case Big-O would be $O(n)$
  - Theoretically, the best case would be where the item is at the start of the array and so would be $O(1)$
  - The average case is theoretically $\frac{n}{2}$, but that is still a Big-O of $O(n)$
  - This works for sorted and unsorted lists
- Binary Search
  - In a binary search, you start by checking the middle of the list
  - If the item is larger than the value in the middle, you check the top half of the list. If it's smaller, check the bottom half
  - Repeat this process, splitting the list in half, until the item is found
  - In the case of an list with an even number of items, look at the middle item rounded down (e.g. in a list of 8 items, start with the item at index 3)
  - The worst-case would be $1 + \log_2 n$ operations, so $O(\log_2 n)$
  - The best-case would still be $O(1)$ if the item is at the mid-point in the list
  - The average would be $O(\log_n 2)$
  - This only works for pre-sorted lists, so if the list is not sorted, there is an efficiency tradeoff to sort the list before it can be searched

Sorting Algorithms

- Selection Sort
  - This works by sorting the list one item at a time
  - It does this by dividing the list into two sections: the sorted part on the left and the unsorted part on the right
  - The smallest or largest item is selected from the list (depending if sorting ascending or descending) and swapped into the first position in the sorted part
  - This process is repeated until the entire list is sorted

  - Each selection requires $n-1$ comparisons, and the selection process must be performed $n-1$ times
  - Therefore both the best and worst-case would be $(n-1)(n-1) = n^2 - 2n + 1$ and therefore a Big-O of $O(n^2)$

- Bubble sort
  - This works by repeatedly looping through the list, swapping adjacent items that are in the wrong order
  - This causes either the largest or smallest item to 'bubble up' to the top of the list
  - The process is repeated until all items have been rearranged into the correct order
  - Each iteration requires $n-1$ comparisons, and must be repeated n times
  - Therefore the best and worse case would be $(n)(n-1) = n^2 - n$ and therefore a Big-O of $O(n^2)$
  - There is a variant of bubble sort which uses a flag to determine when the list is sorted, and so if the list starts out sorted, it would only need n operations and therefore the best-case Big-O becomes $O(n)$

- Insertion Sort
  - This works by sorting the list one item at a time
  - The list is once again divided into a sorted and unsorted section
  - Start by iterating over the items in the array until the first unsorted item is found
  - Then shift it down a place and check again. If it is sorted now, it has been inserted into the correct place. If not, continue shifting and comparing items until it is in the "relatively" correct place
  - This may not be the correct position, but it is now sorted correctly relative to all the items it has been compared against
  - The insertion process should be repeated until the entire list is iterated over without finding an item out-of-order
  - The best-case is that the list is already sorted, and so n comparisons are made to determine that it is sorted, resulting in a best-case Big-O of $O(n)$
  - The worst-case is that the list is entirely unordered, and so n comparisons need to be made n times and therefore a Big-O of $O(n^2)$

Recursion

- With an iterative algorithm, the code is explicitly repeated using a for or while loop
- With a recursive algorithm, the code repeats implicitly, depending upon how many times the function is called within itself
- All recursive algorithms use a divide-and-conquer strategy to split the problem into smaller problems
- However, there are some significant issues with recursive algorithms:
  - If a recursive function does not have a **stopping** or **base** case, it will infinitely recurse
  - There is also system overhead when calling functions, so a recursive function is often less time efficient than an iterative function
  - There is a limit to how many times you can recurse before you get a stack overflow error
  - They can also end up using an excessive amount of memory due to placing everything on the stack

- The following are reasons recursive algorithms may not be ideal:
  - Some algorithms or data structures are not well suited to be iterated recursively
  - The recursive solution may be significantly harder to understand or program than its iterative counterpart
  - There may be unforseen space or time consequences to using a recursive algorithm