Review of Data Structures and Algorithms

COMP3121/9101 23T1

February 10, 2023

Throughout this document, we use $O(\ldots)$ for the time complexity of various operations. In lecture 2, we will see that while this notation is sufficient, $\Theta(\ldots)$ would convey slightly more information.

1 Data structures

Each data structure described below is an *abstract data type*. Any type of data (e.g. integers, floating-point numbers, strings) can be stored in one of these data structures (subject to some restrictions). We describe briefly how the data is organised as well as the main operations supported and their time complexities.

1.1 Array

An array stores n items with consecutive indices. In this course, we use the notation convention that indices are numbered $1, \ldots, n$ (rather than $0, \ldots, n-1$). We'll denote

- the entire array by $A : [3, 1, 2, 1, 9, 1, 0, 1]$,
- a subarray by $A[3..6] : [2, 1, 9, 1]$, and
- an array element by $A[5] = 9$.

In this course, we will only talk about static arrays, i.e. arrays of fixed size.

Array operations:

- Random access takes $O(1)$; given an index i, we can directly access and modify the associated element $A[i]$.
- Insertion or deletion takes $O(n)$ as we need to recreate the array.
- Without special structure, search takes $O(n)$ time in the worst case as we may need to check every element of the array.

1.2 Linked list

A linked list stores items each with a link to the next item. In a doubly linked list, each item also includes a link to the previous item. We will discuss only doubly

linked lists unless specified otherwise, as we aren't concerned by the $2\times$ overhead since it is only a small constant factor.

Linked list operations:

- Accessing the next or previous item takes $O(1)$ by following the relevant link.
- Random access takes $O(n)$ in the worst case, as we need to follow links one at a time until we reach the desired index.
- Insertion and deletion from a designated position both take $O(1)$ as we only need to modify up to four links.
- Search again takes $O(n)$ in the worst case.

1.3 Stack

A stack stores items in LIFO (last in first out) order.

Stack operations:

- Accessing the top of the stack takes $O(1)$.
- Insertion to and deletion from the top of the stack take $O(1)$, but we cannot insert or delete at other positions.

1.4 Queue

A queue stores items in FIFO (first in first out) order.

Queue operations:

- Accessing the front of the queue takes $O(1)$.
- Insertion to the back and deletion from the front of the queue take $O(1)$, but we cannot insert or delete at other positions.

2 Searching algorithms

2.1 Linear search

Given an array A, we can search for a value x by comparing each element of A to x. With early exit, the best case runtime is $O(1)$, but the worst case is $O(n)$.

Without further assumptions about the array, it is impossible to avoid checking each element of A.

Similar methods can be used to find the maximum or minimum value in an array of comparable items, i.e. items that can be compared to each other.

3 Sorting algorithms

3.1 Bubble sort

Algorithm:

1. For each pair of adjacent elements, compare them and swap them if they are out of order (i.e. if $A[i] > A[i+1]$).
2. Repeat until no swaps are performed in an entire pass.

This algorithm is correct because:

- The first pass finds the largest element and stores it in $A[n]$.
- Each subsequent pass fixes the next largest element.
- After at most n passes, the algorithm must terminate, at which point all n elements are correctly positioned.

The time complexity is:

- $O(n)$ in the best case; if the array is already sorted, we can early exit after the first pass.
- $O(n^2)$ in the worst case; if the array is reverse sorted, we will need to complete n passes totalling

$$(n-1) + (n-2) + \ldots + 2 + 1 = \frac{n(n-1)}{2}$$

  comparisons.
- $O(n^2)$ in the average case; it's very rare to be able to fix several elements at once, so in most instances you will need close to n passes.

Bubble sort is generally slow for large arrays, and is fast only if the array is nearly sorted already (i.e. very few passes are needed).

There are hardly any practical applications, but the logic of swapping adjacent elements in order to sort an array is useful in proving the correctness of certain greedy algorithms.

3.2 Selection sort

Algorithm:

1. Perform a linear search to find the smallest element, and swap it with $A[1]$.
2. Repeat on $A[2..n]$, and so on.

This algorithm is clearly correct, again fixing one element per pass.

The time complexity is $O(n^2)$ in all cases, as we must complete n passes of the array totalling $\frac{n(n-1)}{2}$ comparisons regardless of the input array.

While it is intuitive for humans sorting small arrays, e.g. a hand of cards, selection sort is always slow on large arrays when considered in terms of the number of comparisons. It could however be useful on specialised hardware where swaps (i.e. writes) are much more expensive than comparisons, because it is guaranteed to do no more than n swaps.

3.3 Insertion sort

Algorithm:

1. Make an empty linked list B, which will be kept in sorted order.
2. For each element of A:
    - compare it to each element of B from back to front until its correct place is found.
    - insert it at that place.
3. Copy the entries of B back into A.

This algorithm is again correct because it clearly builds the sorted array one item at a time.

Similar to bubble sort, the time complexity is $O(n)$ in the best case (a sorted array), $O(n^2)$ in the worst case (a reverse sorted array) and $O(n^2)$ in the average case. As stated, the algorithm also takes $O(n)$ additional space, but this can be improved to $O(1)$.

Insertion sort is generally slow for large arrays, and is fast only if the array is nearly sorted (i.e. very few comparisons are needed for each element of A).

However, it is often used to sort small arrays as the last step of a more complicated sorting algorithm.