Data Structures

Based on lectures by

Notes taken by yehelip

Winter 2025

These notes are not endorsed by the lecturers. I have revised them outside lectures to incorporate supplementary explanations, clarifications, and material for fun. While I have strived for accuracy, any errors or misinterpretations are most likely mine.

Contents

1 Introduction

1.1 Abstract data types

Example 1.1. How can we make a data structure for 15,000 students that allows insertion and access to students by their id in time complexity of $O(1)$.

Data structures appear everywhere.

Definition 1.1 (Abstract data type)**.** To be added.

Example 1.2 (Stack)**.** A stack has multiple basic fuctionalities.

- `push(S,x)` - add `x` to the top of the stack.
- `top(S)` - access top of the stack. Return null in the stack is empty.
- `pop(S)` - deletes the element on the top of the stack. Does not affect an empty stack.
- `create()` - returns an empty stack.
- `is_empty(S)` returns an empty stack.

We can implement this abstract data structure using either an array, or a list. Which is more efficient?

A clear disadventage of an array is that the size of the stack is limited.

A disadventage of a list is that our memory is limited.

In the real world we use a list of arrays. We define an array of size N to implement the stack, and when we reach the maximum capacity of the array, we link a new array of size N.

Exercise 1.1. What is the absract data type defined by the functions

- `create(c)` - returns an initial instance.
- `read(c)` - returns an integer.
- `read(create(c))` - returns `0`.
- `increment(c)` - changes `c` such that `read(increment(c) = read(c) + 1`.

The surprising answer is a counter!

- `create(c)` - allocates memory to an integer c.
- `read(c)` - returns `c`.
- `increment(c)` - changes `c` such that `c = c + 1`.

But there is a problem since there could be overflow! How can we solve this?

Example 1.3 (Queue)**.** A queue has multiple basic fuctionalities.

- `enqueue(Q,x)` - add x to the end of the queue.
- `head(S)` - access start of the stack. Return null in the queue is empty.
- `dequeue(S)` - deletes the first element of the queue, and return the first element of the modified queue. Does not affect an empty queue.
- `create()` - returns an empty queue.
- `is_empty(`S`)` returns true if the queue is empty.

We can implement a queue using an array, or a linked list.

In `c++` we can implement an abstract data type in multiple ways, then by using

The quality of the implementation is affected by the time and space complexity of the functions, and the simplicity of the implementation.

1.2 Time complexity

Definition 1.2 (Run time)**.** The run time of algorithm A on input x is the number of machine commands that the algorithm runs on the input. We denote it $\text{time}_A(x)$.

Remark 1.1. We don't need to bother about the different in machines that execute the commands, and we also don't bother about the difference in run-time of different commands.

Since we can't possibly write the run time for all inputs, we denote:

$$\text{t}_A(n) = \max\left\{\text{time}_A(x) \mid |x| = n\right\}.$$

Example 1.4 (For loop)**.** Consider the following code:
we have that

$$n + 1 \leq \text{t}_A(n) \leq c_1 \cdot n + c_2.$$

Where c_1, c_2 are constants that depend on the the implementation of the specific programming language.

Example 1.5 (check prime)**.** Consider the following code:
We have that

$$\begin{aligned}
&\text{time}(9888) = c_1 + c_2 \\
&\text{time}(9973) = 9971 \cdot c_1 + c_2 \\
&\text{time}(4) = 9971 \cdot c_1 + c_2 \\
&\text{time}(n) = 10^n \cdot c_1 + c_2
\end{aligned}$$

Remark 1.2. Notice that we talk about the *size* of the input and not the *value* of the input. Here n is the ?????

As we said earlier we don't really care about the constants.
We can think about $f(n)$ the function of the real time of the algorithm.

Definition 1.3. Let $f(n)$, $g(n)$ be two positive functions defined on the natural numbers. We say that $f(n) \in O(g(n))$ if exist $c > 0$ and $n_0 > 0$ such that for all $n > n_0$ we have $f(n) \leq c \cdot g(n)$.

If $f(n) \in O(g(n))$ we usually denote $f(n) = O(g(n))$ and say that $g(n)$ is an asymptotoic bound of $f(n)$.

Proposition 1.1. *Let k be a constant. Then*

$$f(n) = \sum_{i=0}^{k} a_k n^k = O(n^k).$$

Proof. Let $a_{\max} = \max\{1, a_0, \ldots, a_k\}$. Then we have

$$f(n) \leq a_{\max}(1 + n + \cdots + n^k) = a_{\max}\frac{n^{k+1}}{n-1} \leq a_{\max}\frac{n^{k+1}}{n/2} = 2a_{\max}n^k.$$

Thus we can choose $n_0 = 2$ and $c = 2a_{\max}$ and then $f(n) \leq c \cdot n^k$ for all $n > n_0$. □

Here are a couple more examples:

$$\begin{aligned}
&f(n) = 10006 = O(1) \text{ (Constant)} \\
&f(n) = 4n + 27 = O(n) \text{ (Linear)} \\
&f(n) = n\log_2(n) + 3n^2 = O(n^2) \text{ (Polynomial)}
\end{aligned}$$

Proposition 1.2. *For all $a, b > 1$ we have that*

$$f(n) = \log_a(n) = O(\log_b(n)).$$

A polynomial is stronger than any logarithm.

Proof. For all $\epsilon > 0$ and $a > 1$ we have

$$f(n) = \log_a n = \frac{\epsilon \cdot}{a}.$$

□

More examples are

$$f(n) = a^n = O(b^n) \text{ (Exponential)}$$
$$f(n) = n^k + a^n = O(a^n)$$
$$f(n) = n \cdot log_2 n = O(n \log n)$$

We can also show that

$$f(n) = 3^n \neq O(2^n) \text{ (Exponential)}$$
$$f(n) = e^n \neq O(n^k)$$
$$f(n) = \log n \neq O(\log \log n)$$

Proof. Assume by contradiction that exist $c > 0$ and $n_0 \geq 0$ such that for all $n > n_0$ we have $3^n \leq c\dot{2}^n$. This implies that $\left(\frac{3}{2}\right)^n \leq c$ for all $n > n_0$. But this is obviously a contradiction which completes the proof. □

Proof. Let $k > 0$. We know from analysis that

$$\lim_{x \to \infty} \frac{x^k}{e^x} = 0$$

by L'hopital which imlies that $e^n \neq O(n^k)$ for all $k > 0$ as wanted. □

Proof. Similarly we have that

$$\lim_{n \to \infty} \frac{\log \log n}{\log n} = 0$$

which completes the proof. □

Example 1.6. Summing elements of an arary of length n is $O(n)$.

Example 1.7 (Multiplying matrices)**.** When multiplying matrices of order $m \times m$ the complexity of the algorithm is $O(m)$. But m is not the size of the input. Sinc the input is of 2 matrices it is equal $2m^2$. Therefore

$$T(n) = O(m^3) = O\left(\frac{n^{\frac{3}{2}}}{2^{\frac{3}{2}}}\right) = O(n^{\frac{3}{2}})$$

Example 1.8 (Binary search)**.** In a binary search we have that the algorithm executes some commands on input of size n, and then it executes the same commands on input of size

Example 1.9 (Weird loops)**.** Consider the following code: Roughly we have that

$$T(n) \leq n(n-1) = O(n^2).$$

But more precisely we have that