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

Definition 1.4. Let $f(n)$, $g(n)$ be two positive functions defined on the natural numbers. We say that $f(n) \in \Omega(g(n))$ if exist $c > 0$ and $n_0 > 0$ such that for all $n > n_0$ we have $f(n) \geq c \cdot g(n)$.

Remark 1.3.

$$g(n) = O(f(n)) \iff f(n) = \Omega(g(n))$$

Definition 1.5. Let $f(n)$, $g(n)$ be two positive functions defined on the natural numbers. We say that $f(n) = \Theta(g(n))$ if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Example 1.10. We have that $H_n = \Theta(\ln n)$ because

$$\ln(n+1) = \int 1^{n+1}$$

Definition 1.6. Let $f(n)$, $g(n)$ be two positive functions defined on the natural numbers. We say that $f(n) = o(g(n))$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0.$$

Example 1.11.

Note that asymptotic complexity isn't always what we want to optimize.

Example 1.12. Suppose algorithm A has a running time of $\mathrm{t}_A(n) = 100n$, and algorithm B has a running time of $\mathrm{t}_B(n) = 5n \log_2 n$. Asympotically algorithm A is better because it is linear, but for algorithms with $n < 2^{20}$ algorithm B is more efficient.

Example 1.13. We would rather have $\mathrm{t}_A(n) = n^2$ than $\mathrm{t}_B(n) = 10^{160}$ even though $B = O(1)$.

2 Arrays, matrices, sparse matrices and linked lists

2.1 Arrays

Example 2.1 (Array)**.** An array has multiple basic fuctionalities.

- `create(type,I)` - returns an array of elements of type `type` with the index set `I`.
- `store(A,i,e)` - stores `e` of type `type` with index `i` to the array. This usually looks like `A[i] = e`.
- `get(A,i)` - returnes the element in index `i`. This usually looks like `A[i]`.

All the values in the array are of the same type. Notice that we didn't define what happens if we try to store an element of a different type, or what to return when we use get on an uninitialized value.

Here are some examples of implementations.

Example 2.2 (Continuous memory block)**.** This is the most trivial implementation. In this way we have

- The functions `store` and `get` in $O(1)$
- TO BE CN

Example 2.3 (Multiple continuous memory block of size n)**.** To implemet an array of size m we need to allocate $k = \lceil m/n \rceil$ blocks. In this way we only need to keep a pointer array to the start of each block. To access an element in this kind of array, we can use basic arithmetics. For example, if $n = 6$ and $i = 16$ the address of $A[16]$ is

$$base\left(\left\lfloor \frac{16}{6} \right\rfloor\right) + 16\%6 = base\,(2) + 4$$

where *base* is the pointer array to the start of the blocks. The address computation time is still $O(1)$.

Example 2.4 ($2d$-array)**.** In a $2d$-array $A[i][j]$ each row $A[i]$ is a $1d$-array of size n. To find the element in $A[i][j]$ we can calculate

$$base + i \cdot n + j$$

where *base* is the beginning of the array.

Example 2.5 (n-dimensional array)**.** An n-dimensional array is very similar to a 2-dimensional array. To find the address of $A[i_d] \dots [i_1]$ we get

$$base + i_d \cdot n_{d-1} \cdots n_1 + i_{d-1} n_{d-2} \cdots n_1 + \cdots + i_1$$

The number of operations in this calculations is $O(n^2)$.

Question. Is there a way to optimize this calculation.

Answer. Yes! using Horner's method we can do the same computation in a linear amount of operations.

Remark 2.1. In general, when scanning the elements of an n-dimensional array it is better to scan them by rows and not by columns because of *locality*. It is easier for the computer to access memory that is close to the previously accessed memory. Reading memory is more expensive than than calculting an operation in the GPU.

2.2 Initializing an array in $O(1)$

In the trivial way, initializing an array of size n takes $O(n)$ operations. We want to initialze an array in $O(1)$, for this, we will use more memory.

Example 2.6 (First try)**.** We will also allocate an auxillary array of size n. Each time we store a value in index i we add index i to the top of the auxillary array. In this way, before trying to access to array, we check if index i is in the auxillary array, and if not, return 0 (if we wanted to initialize the array in 0).

This implementation is problamatic because when accessing the element in index i we first need to scan the auxillary array which takes $O(n)$ operations.

Example 2.7 (Second try)**.** This time, we will hold three arrays. An array for values, shortcuts, and indexes with value. In this way, when accessing an element in the index i, we first check if the shortcuts array in index i.

To implement this we first define the function

Remark 2.2. After $top \geq n$ we don't have to use the auxillary arrays anymore.

Merits

- Asymptotically this is a great solution.

- Practically

Demertis

- Takes three times more space.
- The initialization time is later backstabbed by the new implementations of `get` and `store` (amortized analysis).

2.3 Matrices

Example 2.8 (Symmetrical matrices)**.** Suppose we have a matrix like

$$\begin{pmatrix} 1 & 2 & 4 & 7 & 12 \\ 2 & 3 & 5 & 8 & 1 \end{pmatrix}$$

We can just hold some
To calculate the address we get

Example 2.9 (Sparse matrix)**.** A sparse matrix is a matrix where most elements are 0 (or some other constant).

Definition 2.1 (Sparse matrix)**.** If $M_1, \ldots, M_n$ is a sequence of matrices where the number of elements different from some constant c is $o(m(n))$ where $m(n)$ is the number of elements in the nth matrix then it is a sequence of sparse matrices.

Example 2.10. Diagonal matrices are sparse matrices.

Remark 2.3. In sparse matrices of order $n \times n$ the number of elements different than c is $o(n^2)$.

One way to represent these matrices is by a lexicographical list of triples $(i, j, A_{i,j})$ that represent the values different than c in the matrix. On one hand, we lose `get` in constant time. On the other hand, in this way allocation is way more efficient. Addition and multiplication of sparse matrices are way more efficient.

Example 2.11 (Addition of sparse matrices)**.** Addition in usual matrices of order $n \times n$ takes $o(n^2)$ time. When adding sparse matrices in this repersentation, adding them is as bad as combining their lists (of elemenets different from c). Suppose their lists are of size m and k, the time complexity is now only $o(m + k)$.

2.4 Linked lists

Recall that a linked list
There are different kinds of linked lists.

Example 2.12 (Dummy node linked list)**.** A dummy node linked list is just like a standard linked list, with an additional node at the beginning without any meaningful value to make deletion more easily

Example 2.13 (Cyclic linked list)**.** A linked list where the last element is pointing to the head node. Some merits are

- We can reach any element from any element.
- We can concatenate lists in constant time, because we have constant time access to the pointer of the last node.

Example 2.14 (Doubly linked list)**.** In a doubly linked list we have not only pointers to the next nodes, but also the the previous nodes. Mertis:

- We can reach any element from any element.
- Given access to a node t we can delete it in constant time.

There is also a tricky deletion in standard linked lists.
TO BE CONTINUED

2.5 Extra

Parse polynomials.

Definition 2.2. DEFINE

Instead of representing sparse polynomials in an array of size of the order of the polynomial, we can represent the nonzero mekadmim in a linked list
MORE IN PRESENTATION