Advanced Data Structures

Amalia Duch

CS-UPC

February 10, 2020

- Become acquainted with the main and classic data structures of central areas of computer science and identify their major properties.
- Become familiar with the mathematical tools usually used to analyze the performance of data structures.
- Examine ideas, analysis and implementation details of data structures in order to assess their fitness to different classes of problems.
- Select, design and implement appropriate data structures to solve given problems.

- Preliminaries.
- Multidimensional and metric data structures, searching in metric spaces, associative retrieval and object representation.
- Geometric and kinetic data structures.
- External memory/Cache oblivious.
- Hashing.
- Heaps.
- Self-adjusting data structures.
- Randomized data structures.
- Strings.
- Miscellaneous.

- Basic knowledge of the C++ programming language.
- Basic knowledge of algorithm analysis methods (in particular asymptotic complexity).
- Basic knowledge of elementary data structures such as stacks, queues, linked lists, trees, and graphs as well as of sorting methods such as insertion sort, heap sort, merge sort, and quick sort.

Grade = 60% FW + 10% SP + 30% AW

- FW = Final Work (graded from 0 to 10) in which each participant is required to present a research paper. The presentation consists of:
  - 3-5 minutes background on the topic of the paper.
  - 1 minute overview of the key ideas of the paper.
  - 5 minutes presentation with most important details.
  - 5 minutes demo of a program that implements the ideas introduced in the paper.
- SP = Summaries and participation (graded from 0 to 10).
- AW = Additional Work: 3 deliveries from these possibilities: quizzes (graded from 0 to 10), one per content's item, implementation and experimentation of a data structure, homeworks, notes of one of the topics, contributions to wikipedia, etc...

What is a data structure?

In computer science, a *data structure is a particular way of storing and organising data in a computer so that it can be used efficiently [Wikipedia].*

In general, a data structure is a kind of higher-level instruction in a virtual machine: when an algorithm needs to execute some operations many times, it is reasonable to identify what exactly the needed operations are and how they can be realised in the most efficient way. This is the basic question of data structures: given a set of operations whose intended behaviour is known, how should we realise that behaviour [Brass].

A data structure models some abstract object. It implements a number of operations on this object, which usually can be classified into

- creation and deletion operations,
- update operations, and
- query operations.

Data structures that allow updates and queries are called *dynamic* data structures, while data structures that are created just once for some given objects and allow queries but no updates are called *static* data structures.

In general, dynamic data structures are preferable, but we also need to discuss static structures because they are useful as building blocks for dynamic structures and because for some of the more complex objects we might encounter, no dynamic structure is known.

We want data structures that are *fast* and, if possible, *small*.
Usual complexity measures are worst-case, average-case and amortised complexity

- Different kinds of data structures are suited to different kinds of applications, and some are highly specialised to specific tasks.
- For example, B-trees are particularly well-suited for implementation of databases, while compiler implementations usually use hash tables to look up identifiers.
- Data structures provide a means to manage huge amounts of data efficiently, such as large databases and Internet indexing services.

Why study data structures?

- Usually, efficient data structures are a key to designing efficient algorithms.
- Some formal design methods and programming languages emphasise data structures, rather than algorithms, as the key organising factor in software design.
- Storing and retrieving can be carried out on data stored in both main memory and in secondary memory.

Algorithms + Data structures = Programs [N. Wirth, Turing award 1984].

”Data dominates. If you’ve chosen the right data structures and organised things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.” [Rob Pike]

Elementary data structures

- Tuples
- Vectors
- Lists
- Stacks
- Queues

- Efficiency of an algorithm = its consumption of computing resources: running time and memory space
- Analysis of algorithms $\rightarrow$ Properties of algorithm's efficiency
  - Compare different algorithmic solutions
  - Predict the amount of resources required by an algorithm or DS (data structure)
  - Improve existent algorithms and DSs and guide the design of new ones.

Generally, given an algorithm A with input data set $\mathcal{A}$ its efficiency or cost (in time, space, number of I/O operations, etc.) is a function T from $\mathcal{A}$ to $\mathbb{N}$ (or $\mathbb{Q}$ or $\mathbb{R}$):

$$\begin{aligned} T : \mathcal{A} &\to \mathbb{N} \\ \alpha &\to T(\alpha) \end{aligned}$$

To characterize function T could be very complicated and could produce unmanageable information, not useful in practice.

Let $\mathcal{A}_n$ be the input set of size n and $T_n : \mathcal{A}_n \to \mathbb{N}$ the function T restricted to $\mathcal{A}_n$.

- *Best case cost*:

$$T_{\text{best}}(n) = \min\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- *Worst case cost*:

$$T_{\text{worst}}(n) = \max\{T_n(\alpha) \mid \alpha \in \mathcal{A}_n\}.$$

- *Average case cost*:

$$\begin{aligned} T_{\text{avg}}(n) &= \sum_{\alpha \in \mathcal{A}_n} \Pr(\alpha)\, T_n(\alpha) \\ &= \sum_{k \geq 0} k \, \Pr(T_n = k). \end{aligned}$$

1. For all $n \geq 0$ and for any $\alpha \in \mathcal{A}_n$

$$T_{\text{best}}(n) \leq T_n(\alpha) \leq T_{\text{worst}}(n).$$

2. For all $n \geq 0$

$$T_{\text{best}}(n) \leq T_{\text{avg}}(n) \leq T_{\text{worst}}(n).$$

An essential characteristic of the cost (worst case, best case, average case) is its growth rate

Example

1. Linear functions: $f(n) = a \cdot n + b \Rightarrow f(2n) \approx 2 \cdot f(n)$
2. Quadratic functions:
   $q(n) = a \cdot n^2 + b \cdot n + c \Rightarrow q(2n) \approx 4 \cdot q(n)$

Linear and quadratic functions have different growth rates, or, different orders of magnitude.

$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	262144
5	32	160	1024	32768	$6.87 \cdot 10^{10}$
6	64	384	4096	262144	$4.72 \cdot 10^{21}$
			$\dots$		
ℓ	N	L	C	Q	E
$\ell + 1$	$2N$	$2(L + N)$	$4C$	$8Q$	E^2

Constant factors and lower order terms are irrelevant regarding growth rate. For instance, $30n^2 + \sqrt{n}$ has the same growth rate than $2n^2 + 10n \Rightarrow$ asymptotic notation.

Definition

Given a function $f : \mathbb{R}^+ \to \mathbb{R}^+$ the class $\mathcal{O}(f)$ (big O of f) is

$$\mathcal{O}(f) = \{g : \mathbb{R}^+ \to \mathbb{R}^+ \mid \exists n_0 \, \exists c \, \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

And it reads, a function g is in $\mathcal{O}(f)$ if there is a constant c such that $g < c \cdot f$ for all n greater than n_0.

Although $\mathcal{O}(f)$ is a set of functions, traditionally we write $g = \mathcal{O}(f)$ instead of $g \in \mathcal{O}(f)$. Nevertheless, to write $\mathcal{O}(f) = g$ has no sense.

Basic properties of $\mathcal{O}$:

1. If $\lim_{n\to\infty} g(n)/f(n) < +\infty$ then $g = \mathcal{O}(f)$
2. Reflexivity: for every function $f : \mathbb{R}^+ \to \mathbb{R}^+$, $f = \mathcal{O}(f)$
3. Transitivity: if $f = \mathcal{O}(g)$ y $g = \mathcal{O}(h)$ then $f = \mathcal{O}(h)$
4. For every constant $c > 0$, $\mathcal{O}(f) = \mathcal{O}(c \cdot f)$

Other useful asymptotic notations are: Ω (omega) and Θ (zita). The first one defines a set of functions bounded below by a given function:

$$\Omega(f)=\{g:\mathbb{R}^+\to\mathbb{R}^+ \mid \exists n_0\ \exists c>0\ \forall n\geq n_0 : g(n)\geq c\cdot f(n)\}$$

Ω is reflexive and transitive; if $\lim_{n\to\infty} g(n)/f(n) > 0$ then $g = \Omega(f)$. Besides, if $f = \mathcal{O}(g)$ then $g = \Omega(f)$ and viceversa.

We say that $\mathcal{O}(f)$ is the class of functions that don't grow faster than f. Analogously, $\Omega(f)$ is the class of functions that don't grow slower than f.
Finally,

$$\Theta(f) = \Omega(f) \cap \mathcal{O}(f)$$

is the class of functions with the same rate growth than f.
Θ is also reflexive and transitive. Furthermore, it is symmetric: $f = \Theta(g)$ if and only if $g = \Theta(f)$. If $\lim_{n\to\infty} g(n)/f(n) = c$ with $0 < c < \infty$ then $g = \Theta(f)$.

Additional properties of asymptotic notations (inclusions are strict):

1. For any constants $\alpha < \beta$, if f is a growing function then $\mathcal{O}(f^{\alpha}) \subset \mathcal{O}(f^{\beta})$.
2. For any constants a y $b > 0$, if f is growing, $\mathcal{O}((\log f)^{a}) \subset \mathcal{O}(f^{b})$.
3. For any constant $c > 1$, if f is growing, $\mathcal{O}(f) \subset \mathcal{O}(c^{f})$.

Conventional operators (sums, substractions, divisions, etc.) over classes of functions defined by means of asymptotic notations are extended as follows:

$$A \otimes B = \{h \,|\, \exists f \in A \wedge \exists g \in B : h = f \otimes g\},$$

where A and B are sets of functions.
Expressions of the form $f \otimes A$ where f is a function is extended as $\{f\} \otimes A$.
This way we can write easily expressions as $n + \mathcal{O}(\log n)$, $n^{\mathcal{O}(1)}$, ó $\Theta(1) + \mathcal{O}(1/n)$.

Rule of Sums:

$$\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max\{f, g\}).$$

Rule of Products:

$$\Theta(f) \cdot \Theta(g) = \Theta(f \cdot g).$$

Similar rules are valid for notations $\mathcal{O}$ and Ω.

Analysis of iterative algorithms

1. The cost of an elemental operation is $\Theta(1)$.
2. If the cost of a fragment S_1 is f and of S_2 is g then the cost of $S_1; S_2$ is $f + g$.
3. If the cost of S_1 is f, of S_2 is g and the cost of evaluate B is h then the worst case cost of:

```
if (B) { S1;}
else {S2;}
```

is $\max\{f + h, g + h\}$.

4. If the cost of S during the i-th iteration is f_i, the cost of evaluate B is h_i and the number of iterations is g then the cost T of

```
while (B) {
    S;
}
```

is

$$T(n) = \sum_{i=1}^{i=g(n)} f_i(n) + h_i(n).$$

If $f = \max\{f_i + h_i\}$ then $T = \mathcal{O}(f \cdot g)$.

Analysis of recursive algorithms

The cost (in worst case, average, ...) of a recursive algorithm $T(n)$ satisfies a recurrent equation: this is, $T(n)$ will depend on the value of T for smaller values of n. Frequently, the recurrence has one of the following forms:

$$T(n) = a \cdot T(n - c) + g(n),$$
$$T(n) = a \cdot T(n/b) + g(n).$$

The first one corresponds to algorithms that have a non recursive part with cost $g(n)$ and do a recursive calls with subproblems of size $n - c$, with c a constant.
The second one corresponds to algorithms with a non recursive part of cost $g(n)$ that do a recursive calls with subproblems of size (approximately) n/b, with $b > 1$.

Theorem

Let $T(n)$ be the cost (worst case, average case, ...) of a recursive algorithm that satisfies the recurrence:

$$T(n) = \begin{cases} f(n) & \text{if } 0 \leq n < n_0 \\ a \cdot T(n-c) + g(n) & \text{if } n \geq n_0, \end{cases}$$

where n_0 is a constant, $c \geq 1$, $f(n)$ is an arbitrary function and $g(n) = \Theta(n^k)$ for a constant $k \geq 0$.
Then

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } a < 1 \\ \Theta(n^{k+1}) & \text{if } a = 1 \\ \Theta(a^{n/c}) & \text{if } a > 1. \end{cases}$$

Theorem

Let $T(n)$ be the cost (worst case, average case, ...) of a recursive algorithm that satisfies the recurrence:

$$T(n) = \begin{cases} f(n) & \text{if } 0 \leq n < n_0 \\ a \cdot T(n/b) + g(n) & \text{if } n \geq n_0, \end{cases}$$

where n_0 is a constant, $b > 1$, $f(n)$ is an arbitrary function and $g(n) = \Theta(n^k)$ for a constant $k \geq 0$.
Let $\alpha = \log_b a$. Then

$$T(n) = \begin{cases} \Theta(n^k) & \text{if } \alpha < k \\ \Theta(n^k \log n) & \text{if } \alpha = k \\ \Theta(n^\alpha) & \text{if } \alpha > k. \end{cases}$$


T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein.
Introduction to Algorithms, Third Edition.
The MIT Press, 2009.


D. E. Knuth.
The Art of Computer Programming: Sorting and Searching, volume 3.
Addison–Wesley, 2nd edition, 1998.