Common data structures

Implementation and complexity

What’s a data structure?

(Wikipedia) “a collection of **data values**, the **relationships** among them, and the functions or **operations** that can be applied to the data”

- an **abstraction** of data that is easy for a programmer to work with
- contains more than data: the data is **organized** in a specific way
- well-defined **operations** can be applied to the data

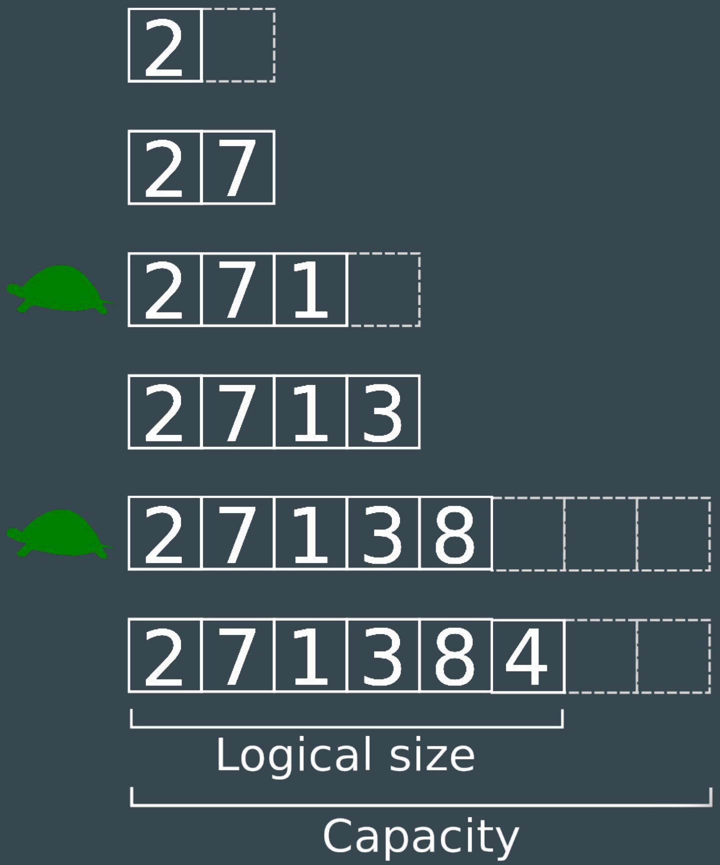
→ it is important to know what data structures exist and which operations can be applied on them

Dynamic arrays (Python lists)

Goal: quickly **access indexed items** in a container and **append new ones** (or remove the last one)

Implementation:

- the language uses more room than needed
- while there is room, appending costs nothing
- when there is no more room, create a new array with more room and copy everything



Dynamic arrays (Python lists)

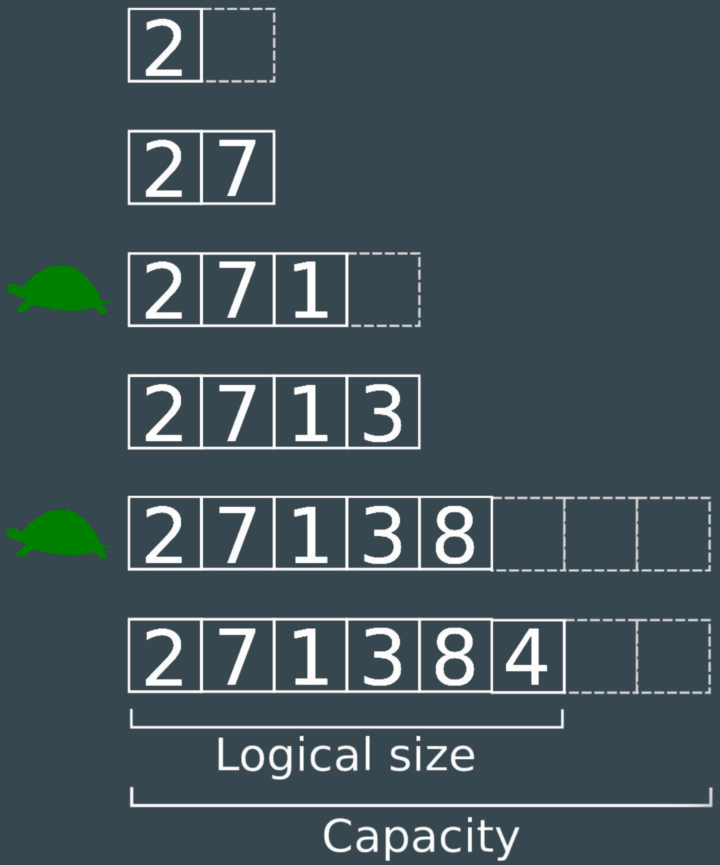
What's the complexity of adding a new item?

if you reach the capacity c_1 and extend the size to $c_2 = 2 * c_1$, the cost is c_1 only once but then the cost will be const for the next c_1 *append* operations

→ on average, the cost is O(1)

What's the complexity of accessing an item?

O(1), position in RAM deduced from its index



Dynamic arrays: in Python and C++

Common operations:	Python	Complexity
Access the i-th item	`arr[i]`	`O(1)`
Add v at the end	`arr.append(v)`	`O(1) avg` `O(n) max`
Insert v at position i	`arr.insert(i, v)`	`O(n)`
Find the position of v	`arr.index(v)`	`O(n)`

When to use:

There are all kinds of use cases. If you often need to perform operations that are not O(1), check if another data structure matches your needs

Quick implementation cheat sheet - Python array

	Instruction	Complexity
Create a new one	`L = list(),` `L = []`	`O(1)`
Access the i-th item	`arr[i]`	`O(1)`
Add v at the end	`arr.append(v)`	`O(1) avg` `O(n) max`
Insert v at position i	`arr.insert(i, v)`	`O(n)`
Find the position of v	`arr.index(v)`	`O(n)`

Quick implementation cheat sheet - Python array

	Instruction	Complexity
Remove the first item with value x	`arr.remove(x)`	`O(n)`
Remove the item at the i-th position	`arr.pop(i)`	`O(n)`
Clear the array	`arr.clear()`	`O(n)`
Get the number of items	`arr.len()`	`O(1)`
Sort	`arr.sort(key=function)`	`O(n*log(n))`
Count the number of items with value x	`arr.count(x)`	`O(n)`

Stacks & Queues

Queues

Stacks

Quick implementation cheat sheet - Python deque

Python: `deque` in `collections` : an implementation of double-ended queue

	Python	Complexity
Create a deque	`dq = deque()`	O(1)
Access the i-th element	`dq[i]`	O(n)
Add v at the head	`dq.appendleft(v)`	O(1)
Remove the head and put it in v	`v = dq.popleft()`	O(1)
Add v at the tail	`dq.append(v)`	O(1)
Remove the tail and put it in v	`v = dq.pop()`	O(1)

Double-ended queues - usage

When to use:

Whenever you need a **queue**

=> When you need to add and delete element often (from head or tail)

When not to use:

- if you want to access very often elements in the middle of the sequence
- you don't need it when inserting and removing elements only on one side (e.g a a **stack**). Just don't do that on the left side

Dictionaries / hash maps

Goals:

- associate **values** to **keys**
- retrieve the value associated to a key in O(1) time

→ unlike real-life dictionaries, these ones are **not ordered**

(in real life, finding a word in a dictionary is O(log n)
unless you have forgotten the alphabetical order)

Dictionaries / hash maps

The underlying structure is called a **hash table**

- a hash function turns the keys into an index
- the keys and values are stored in the data structure based on this index
- multiple strategies exist to manage collisions (cf next slides)(or not)

Example of hash function: sum of the ASCII codes for a string, modulo 42

“hello” → 28 “world” → 6 “INSAlgo” → 33

Dictionaries / hash maps - separate chaining

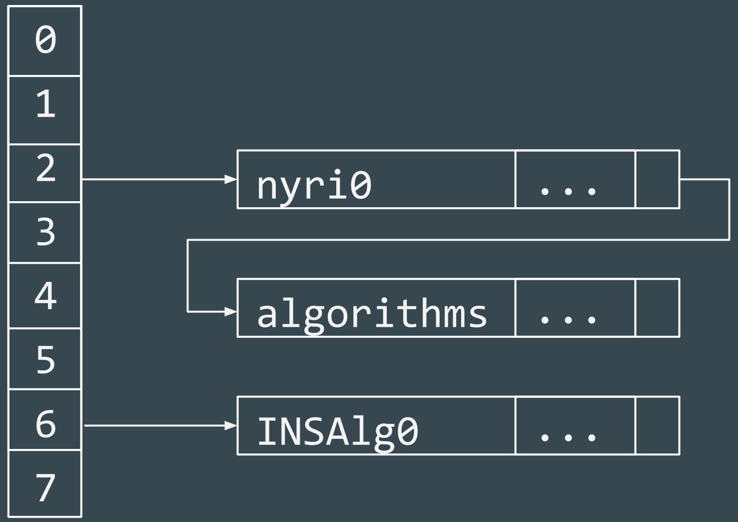
(key, value) couples where keys have the same hash are stored in a common data structure

These structures are kept small enough to have O(1) time access. When too filled, the table is re-created bigger

Often used:

- linked lists
- trees

With sum of ASCII codes modulo 8, and using linked lists:



Dictionaries / hash maps - Open adressing

- No other data structure behind the array
- Collisions are solved with *probing*
- When the fill rate becomes too big, copy the data in a bigger hash table

Linear probing = put at next cell available

Randomized probing: follow a random sequence which seed is given by the hash

→ used in Python

0			
1			
2	2	nyri0	...
3	2	algorithms	...
4			
5			
6	6	INSAlg0	...
7			

(sum of ASCII codes modulo 8, open adressing with linear probing)

Dictionaries / hash maps - in Python and C++

Python: open adressing with randomized probing (congruential RNG)
initial size 8, resized when ⅔ full

(☉_◎) ????

Common operations:

	Python	Complexity
Find or set the value associated to `key`	`dic[key]`	`O(1) avg` `O(n) max`
Check if `key` exists in the dictionary	`key in dic`	`O(1) avg` `O(n) max`

Dictionaries / hash maps - in Python and C++

When to use:

Whenever you need to associate a value to a key.

In Python, the ease of use and high performance make the dictionaries a **very powerful tool** .

When not to use:

When your keys are 0, 1, ..., n
You're better than that.

Quick implementation cheat sheet - Python dict

Python: `dict, built in.`

`Interesting variation : defaultdict in collections`

	Python	Complexity
Create a dict	`d = dict(),` `d = {}`	O(1)
Access an element	`d[key]`	Pseudo O(1)
Put an element	`d[key] = value`	Pseudo O(1)
Find if a key is in dict	`key in d`	Pseudo O(1)
Remove a key	`del d[key]`	Pseudo O(1)

Sets

Goals:

- store unique values
- quickly check if a value is in the set or not

Two common implementations:

- tree-based sets are ordered
- hash sets are faster but unordered
  (they're basically hash tables without values)

Sets - in Python and C++

Python: `set` is a hash set

C++: `unordered_set` is a hash table, `set` a tree

Common operations:

	Python	C++ `set` and `unordered_set`	Complexity	Complexity - C++ `set`
Add the value v to the set	`s.add(v)`	`s.emplace(v)`	`O(1) avg` `O(n) max`	`O(log n)`
Remove the value v from the set	`s.remove(v)`	`s.erase(v)`	`O(1) avg` `O(n) max`	`O(log n)`
Check if v is in the set	`v in s`	`s.find(v)`	`O(1) avg` `O(n) max`	`O(log n)`

Quick implementation cheat sheet - Python set

Python: `set, built in` : implemented as a hash set

	Python	Complexity
Create a set	`S = set()`	O(1)
Access a specific element	`IMPOSSIBLE`	
Put an element	`s.add(element)`	Pseudo O(1)
Find if an element is in a set	`Element in s`	Pseudo O(1)
Remove an element	`s.remove(element)`	Pseudo O(1)

Set arithmetics in Python

<table>
<tr><th></th><th></th><th>Average complexity</th></tr>
<tr><td>s1 <= s2, s1 < s2</td><td>check if s1 is a [proper] subset of s2</td><td>O(n1)</td></tr>
<tr><td>s1 >= s2, s1 > s2</td><td>check if s1 is a [proper] superset of s2</td><td>O(n2)</td></tr>
<tr><td>s1 | s2 | … | sk</td><td>union of s1, s2, …, sk</td><td>O(n1 + n2 + … + sk)</td></tr>
<tr><td>s1 & s2 & … & sk</td><td>intersection of s1, s2, …, sk</td><td>O(min(n1, n2, …, sk))</td></tr>
<tr><td>s1 - s2</td><td>all elements of s1 that are not in s2</td><td>O(n1)</td></tr>
<tr><td>s1 ^ s2</td><td>all elements of s1 or s2 but not both</td><td>O(n1 + n2)</td></tr>
</table>

Sets too are cool :)

Sets - in Python and C++

When to use:

- to mark values already seen
- to keep a collection of unordered unique elements

When not to use:

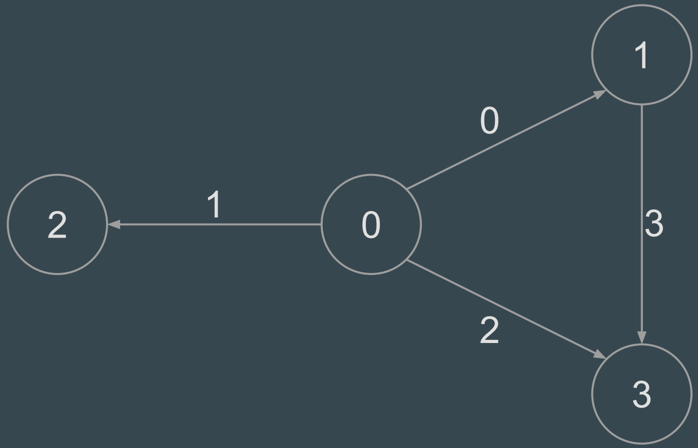
- to make coffee (you just can't)
- to keep twice the same element (really, you just can't)

Bonus : data structures to store Graphs

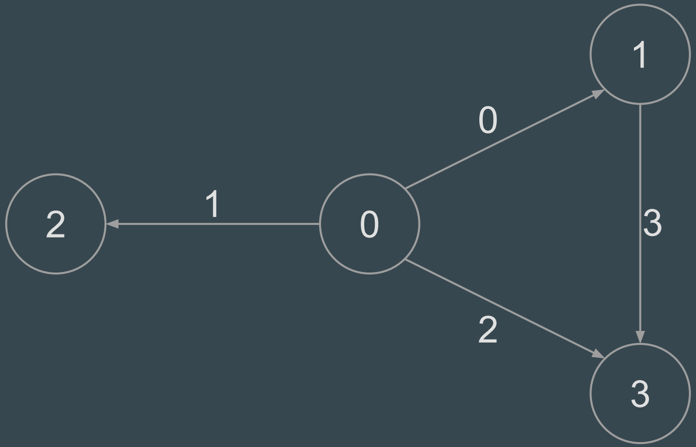
Two main possibilities:

- Adjacency matrix
- Adjacency list

Let's see the result on this graph :



Adjacency Matrix



A matrix where each element a_{ij} give the cost of the edge between node i and node j.

The value of a_{ij} = -1 if there is no edge between i and j

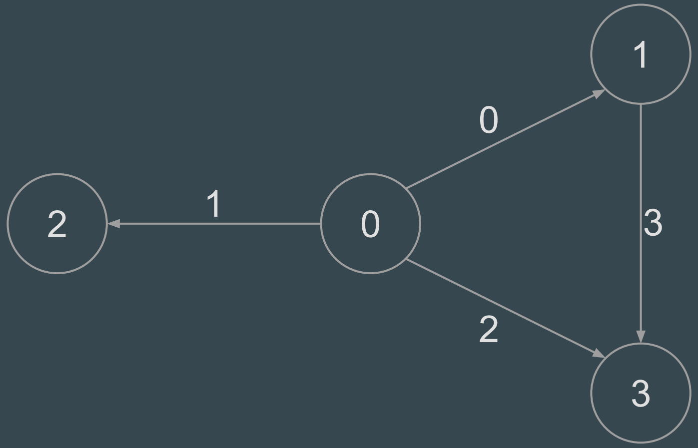
For the example graph, we have:

$$M = \begin{pmatrix} -1 & 0 & 1 & 2 \\ -1 & -1 & -1 & 3 \\ -1 & -1 & -1 & -1 \\ -1 & -1 & -1 & -1 \end{pmatrix}$$

When to use it ?

When we need to check if two random nodes are linked

Adjacency List



For each node, give the list of all adjacent nodes (and the weight of the edge linking them)

Can be implemented with lists or dictionary depending on the indexing of the nodes

For the example we get :

0	1	2	3
[(2,1), (1,0), (3,2)]	[(3,3)]	[ ]	[ ]

When to use it ?

When we need to know the nodes reachable from a current node

Python - `python.collections` classes

`A default Python module`

- `collections.defaultdict(default_factory)` : if an element doesn't exist, create it using the `default_factory` function
- `collections.orderedDict` : a dict which remembers the order of creation of its elements
- `collections.deque` : an implementation of a double-ended queue
- `collections.Counter` : counts occurrences in a list

To be continued...

You probably want to hear about red-black trees, heaps (priority queues), etc...

Any questions ?

Slides: Louis Sugy, Arthur Tondereau for INSAlgo.
Updated by Louis Gombert and Sebastien Goll
Schema of dynamic arrays: Wikipedia

Useful links:

- https://docs.python.org/3/tutorial/datastructures.html (data structures overview)
- https://wiki.python.org/moin/TimeComplexity (Time complexity for most operation)
- https://docs.python.org/3/library/collections.html (Collections library)