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

2		

2	7

2	7	1	

2	7	1	3

2	7	1	3	8			

2	7	1	3	8	4		

Logical size

Capacity

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

$\rightarrow$ on average, the cost is O(1)

What's the complexity of accessing an item?

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

2		

2	7

2	7	1	

2	7	1	3

2	7	1	3	8			

2	7	1	3	8	4		

Logical size

Capacity

Dynamic arrays: in Python and C++

Common operations:

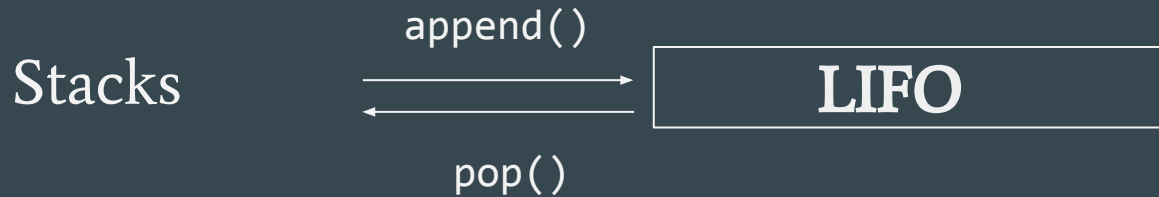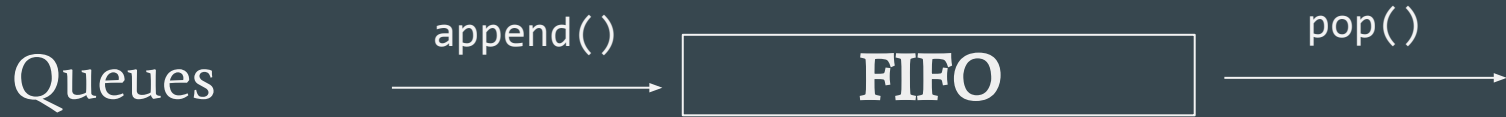|	Python	C++	Complexity
Access the i-th item	`arr[i]`	`arr[i]`	`O(1)`
Add v at the end	`arr.append(v)`	`arr.push_back(v)`	`O(1) avg` `O(n) max`
Insert v at position i	`arr.insert(i, v)`	`arr.insert(i, v)`	`O(n)`
Find the position of v	`arr.index(v)`	`std::find(...)`	`O(n)`

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)`

Stacks & Queues

Queues

append()

→

┌─────────────────────┐
│   **FIFO**   │
└─────────────────────┘

pop()

→

Stacks

append()

→

┌─────────────────────┐
│   **LIFO**   │
└─────────────────────┘

←

pop()

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** (e.g in *breadth-first search* algorithm)

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)

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

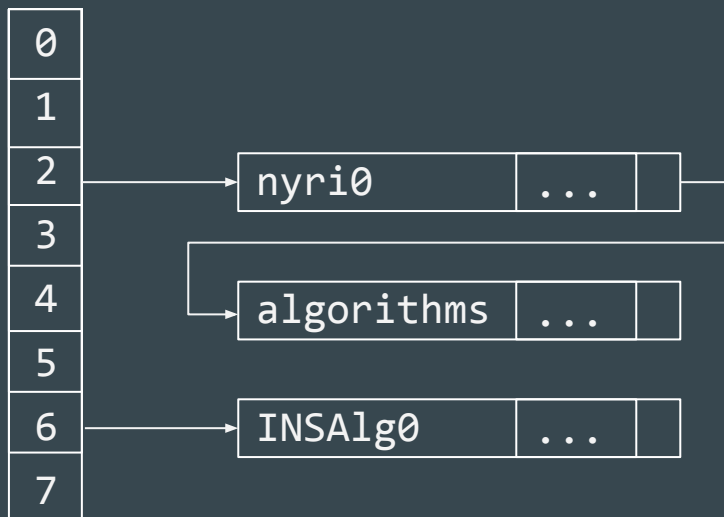(key, value) couples where keys have the same hash are stored in a common data structure

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

C++:   `unordered_map` is a hash table, `map` a red-black tree
          (we will talk about trees later in the year)

Common operations:

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

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

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)

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…

Well, see you in 2019! … this was last year, see you in 2020 perhaps ;)…

2020 note : see you in 2021 if the world hasn't collapsed !

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