Data Structures (Not UML)

Pseudo-code Syntax, Overview and Conclusion

Amandine Decker & Marie Cousin

M1 TAL/SC 2023–2024

Université de Lorraine, LORIA

Class Organisation

Third Lab :

- Reminder : optional exercise TD4 (due December 22nd) ;
- hand back on a sheet of paper (December 19th), send it by e-mail, or hand back on Arche ;
- Correction of the fourth lab (except bonus exercise) is available on Arche ;
- Points and feedback on the optional exercise of TD3 will be on Arche before Wednesday.

Exam :

- January 23rd, from 2pm to 4pm ;
- Closed book exam, except one A4 paper sheet per person ;
- Write in the language you want (French or English) ;
- End grade = exam grade + sum of bonus exercises points.
- If you have "tiers temps", please send us an email before january.

TD3 Optional Exercise

TD3 Optional Exercise : General Feedback

- Do not mix a tree and its root :
  - **add(tree[‘left child’], Q)** adds the whole left child (*i.e.*, the whole sub-tree) to Q ;
  - **add(tree[‘left child’][root’], Q)** adds only the root of the left child (*i.e.*, a single value) to Q ;
  - you can’t compare a tree’s root to a tree’s right child (*i.e.*, a sub-tree), but you can compare it to a tree’s right child’s root.

TD3 Optional Exercise : General Feedback

- Do not mix a tree and its root :
  - **add(tree[‘left child’], Q)** adds the whole left child (*i.e.*, the whole sub-tree) to Q ;
  - **add(tree[‘left child’][root’], Q)** adds only the root of the left child (*i.e.*, a single value) to Q ;
  - you can’t compare a tree’s root to a tree’s right child (*i.e.*, a sub-tree), but you can compare it to a tree’s right child’s root.
- Do not mix queues (FIFO) and stacks (LIFO) ;

TD3 Optional Exercise : General Feedback

- Do not mix a tree and its root :
  - **add(tree[‘left child’], Q)** adds the whole left child (*i.e.*, the whole sub-tree) to Q ;
  - **add(tree[‘left child’][root’], Q)** adds only the root of the left child (*i.e.*, a single value) to Q ;
  - you can’t compare a tree’s root to a tree’s right child (*i.e.*, a sub-tree), but you can compare it to a tree’s right child’s root.
- Do not mix queues (FIFO) and stacks (LIFO) ;
- “What does this algorithm do ?” : we expect the idea of the algorithm, not a translation of the instruction in natural language ;

- Do not mix a tree and its root :
  - **add(tree[‘left child’], Q)** adds the whole left child (*i.e.*, the whole sub-tree) to Q ;
  - **add(tree[‘left child’][root’], Q)** adds only the root of the left child (*i.e.*, a single value) to Q ;
  - you can’t compare a tree’s root to a tree’s right child (*i.e.*, a sub-tree), but you can compare it to a tree’s right child’s root.
- Do not mix queues (FIFO) and stacks (LIFO) ;
- “What does this algorithm do ?” : we expect the idea of the algorithm, not a translation of the instruction in natural language ;
- When adding trees to a queue (or a stack), pay attention to the wanted order, and do not do it “nicely”. For example, in the question 1.a., one of the state of Q is $Q = \left| \begin{tikzcd}[sep=small] 4 \arrow[d] \\ 1 \end{tikzcd}, \begin{tikzcd}[sep=small] & 12 \arrow[dl] \arrow[dr] & \\ 5 & & 14 \end{tikzcd} \right|$, and the next one is $Q = \left| \begin{tikzcd}[sep=small] & 12 \arrow[dl] \arrow[dr] & \\ 5 & & 14 \end{tikzcd}, \begin{tikzcd}[sep=small] & 1 \arrow[dl] \arrow[dr] & \\ \{\} & & \{\} \end{tikzcd} \right|$, and not $Q = \left| \begin{tikzcd}[sep=small] & 1 \arrow[dl] \arrow[dr] & \\ \{\} & & \{\} \end{tikzcd}, \begin{tikzcd}[sep=small] & 12 \arrow[dl] \arrow[dr] & \\ 5 & & 14 \end{tikzcd} \right|$

Syntax

Structures we have seen so far

Structure	Accessibility	Iterable ?
Queue	First element only (FIFO)	No iteration
Stack	Last element only (LIFO)	No iteration
List	Any element by ID	Iteration over elements or by IDs
Dict.	Any element by key	Iteration without order
Tree	Access root, left child, right child	No iteration (traversal algorithms)
Graph	node, nodes directly accessible from it, nodes from which it is directly accessible and the related edges	Informal iteration "for each node of the graph ..."

Variable

Variable

A variable is an object, or a buffer. You can create it, initialise it, modify it. It is used to store data, like a result for example.

Example

$counter \leftarrow 0$

$s \leftarrow "Number\ of\ apples:\ "$

...

$res \leftarrow s + str(counter)$

- Create and initialise a variable ;
- There exists variables of any type ;
- Modify a variable ;

If Block

If/Then/Else block

If the condition is satisfied, instructions 1, 2, etc. are executed. If the condition is not satisfied, instructions A, B, etc. are executed.
Note : It is not mandatory to have the "else" part ; in that case, when the condition is not satisfied, no instructions are executed.

Example

$i \leftarrow 10$
...
if i $<$ 6 **then**
    print(True)
    $i \leftarrow i + 1$
else
    print(False)
end if

- The else part is optional ;
- The end if instruction is mandatory ;
- Do not forget to use indentation ;
- The variable i is NOT created by the block, we need to create it by hand to use it ;
- The block does not change the value of variable i except if a specific instruction to do so is written ;

Loops (1)

For loop

In a *for loop*, we create a variable that will take several values one after the other, *i.e.,* we will iterate over some elements and the variable will take one value at a time. The loop is thus executed a known definite number of times.

While loop

The *loop condition* is a condition that is True or False, it may use some variable used elsewhere in the algorithm. The instructions 1, 2, etc. will be executed as long as the *loop condition* remains satisfied (i.e., True).

Loops (1)

For loop

```
for i in range(0,5) do
    print(i)
end for
```

- The variable i is created by the loop and can be used inside it ;
- It is also incremented by the loop, we do not need to change it by hand.

While loop

```
i ← 0
while i < 5 do
    print(i)
    i ← i + 1
end while
```

- The variable i is NOT created by the loop, we need to create it by hand to use it ;
- It is NOT incremented by the loop, we must change it by hand if we want it to change.

Loops (2)

Nested loops

When you use a loop inside another loop, they are NOT executed in parallel. The *inside* loop will be executed at every step of the *outside* loop.

```
for i in range(0, 3) do
    for j in range(0, 5) do
        print(j)
    end for
end for
```

This algorithm will print 0, then 1, then 2, then 3, then 4 (j loop) and repeat this three times in total (because of i loop).

Loops (2)

Nested loops

When you use a loop inside another loop, they are NOT executed in parallel. The *inside* loop will be executed at every step of the *outside* loop.

$L \leftarrow [0, 1, 2]$
while !*isEmpty*(L) **do**
    for x *in* L **do**
        print(x)
    end for
end while

This algorithm will print 0, then 1, then 2 (For loop) and repeat this an infinite number of time because the condition of the While loop will remain *True*.

Return

Return instruction

This instruction returns *result*, and ends the algorithm.
*(Note : If you encounter a **return** in the middle of an algorithm, **THE FOLLOWING INSTRUCTIONS WILL NOT BE EXECUTED !**)*

```
if condition then
    print("condition True")
    return result
else
    print("condition False")
end if
print("If block finished")
```

- If *condition* is *True*, the algorithm will print "*condition True*", return *result*, and stop ;
- If *condition* is *False*, the algorithm will print "*condition False*", get out of the If block and print "*If block finished*".

Note : If you want to return more than one thing, use comas !

```
return result1, result2, result3
```

Require

Require instruction

This instruction requires the user to give a certain input to your algorithm. The input is stored in a variable that can be used in the algorithm.

- *Comment your instruction to explicit the input you want. For example, if your algorithm requires a list, you can write* "`Require: L` – *We need a list as input*" ;
- *If you need to require several objects, separate the variables with a comma.*

Require

Require instruction

This instruction requires the user to give a certain input to your algorithm. The input is stored in a variable that can be used in the algorithm.

- *Comment your instruction to explicit the input you want. For example, if your algorithm requires a list, you can write* "`Require: L` – *We need a list as input*" ;
- *If you need to require several objects, separate the variables with a comma.*

When requiring an input DO NOT :

- Create an arbitrary object :

1: $L \leftarrow [1, 2, 3, 4, 5, 6]$ `-- Use Require: L instead`

Require instruction

This instruction requires the user to give a certain input to your algorithm. The input is stored in a variable that can be used in the algorithm.

- *Comment your instruction to explicit the input you want. For example, if your algorithm requires a list, you can write* "`Require: L` – *We need a list as input*" ;
- *If you need to require several objects, separate the variables with a comma.*

When requiring an input DO NOT :

- Re-use your variable to store something else / Overwrite your variable :

  Require: L -- *We require a list L*

  1: $L \leftarrow []$ -- *This erases the values given in the input*

Require

Require instruction

This instruction requires the user to give a certain input to your algorithm. The input is stored in a variable that can be used in the algorithm.

- *Comment your instruction to explicit the input you want. For example, if your algorithm requires a list, you can write* "`Require: L` – *We need a list as input*" ;
- *If you need to require several objects, separate the variables with a comma.*

When requiring an input DO NOT :

- Use Require AND create the object afterwards :

  Require: L `-- We require a list L`

  1: $L \leftarrow [1, 2, 3, 4, 5, 6]$ `-- This erases the values given in the input, use a comment to say that L must be a list`

Require instruction

This instruction requires the user to give a certain input to your algorithm. The input is stored in a variable that can be used in the algorithm.

- *Comment your instruction to explicit the input you want. For example, if your algorithm requires a list, you can write* "`Require:` `L` – *We need a list as input*" ;
- *If you need to require several objects, separate the variables with a comma.*

Difference between **Require** and **Creating a new variable** for later :

Require: L `-- We require a list of integers L`

1: $L\_even \leftarrow []$ `-- We create an empty list to store all the even numbers of L`

Data Structures

Queues - Recap (1)

“I want to eat some cupcakes !”

Queue

A queue is a structure containing some objects, organised one after the other. It uses the FIFO principle : First In, First Out (the first object to enter the queue will be the first to leave the queue).

Queues - Recap (2)

Operations on queues

- **add** an element at the end of the queue ;
- **remove** the first element of the queue ;
- **get** the first element of the queue ;
- **check** if the queue is empty ;
- a queue has a length ;

Notation : Q_0 is the empty queue.

The comments give the results of the algorithms when applied to Q1 = |5,1,9|.

- To check if a queue is empty :

  Require: Q `-- if Q1`
  1: **return** $isEmpty(Q)$ `-- False`

- To get an element from queue :

  Require: Q `-- if Q1`
  1: $elt \leftarrow get(Q)$ `-- elt=5`

- To add an element to a queue :

  Require: Q `-- if Q1`
  1: $add(8, Q)$ `-- Q1=|5,1,9,8|`

- To remove an element of a queue :

  Require: Q `-- if Q1`
  1: $remove(Q)$ `-- Q1=|1,9,8|`

Queues - Pay Attention

FIFO

Queues are FIFO = First In First Out (do not mix with stacks, that are LIFO)

The empty queue Q_0

Q_0 is the empty queue. It is a notation we use. When writing "$Q \leftarrow Q_0$", it means that Q is a queue (since Q_0 is) and that Q is now empty.

$Q1 \leftarrow Q_0$
$add(5, Q1)$
$Q_0 \leftarrow Q1$

This algorithm will return an error : you can't assign a (non empty) value to a fix empty structure.

$S \leftarrow Q_0$
$L \leftarrow Q_0$
$add(5, S)$
$L \leftarrow S$

In this algorithm, S and L are both queues because Q_0 is a queue.

Queues - Pay Attention

No iteration

You can't iterate on a queue. It is **NOT AN ITERABLE STRUCTURE**.

Require: Q

$elt \leftarrow Q[5]$

This algorithm returns an error ! A queue is not iterable.

Require: Q

for *elt in* Q **do**

...

end for

This algorithm returns an error ! A queue is not iterable.

Queues - Pay Attention

FIFO

Queues are FIFO = First In First Out (do not mix with stacks, that are LIFO)

The empty queue Q_0

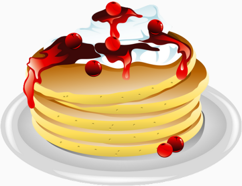
Q_0 is the empty queue. It is a notation we use. When writing "$Q \leftarrow Q_0$", it means that Q is a queue (since Q_0 is) and that Q is now empty.

No iteration

You can't iterate on a queue. It is **NOT AN ITERABLE STRUCTURE**.

Stacks - Recap (1)

"I want to eat some pancakes !"



http ://miam-images.centerblog.net

Stack

A stack is a structure containing some objects, organised one on top of the other. It uses the LIFO principle : Last In, First Out (the last object to enter the stack will be the first to leave the stack).

Stacks - Recap (2)

Operations on stacks

- **add** an element on top of the stack ;
- **remove** the element on top of the stack ;
- **get** the last (= top) element of the stack ;
- **check** if the stack is empty ;
- a stack has a length.

Notation : S_0 is the empty stack.

The comments give the results of the algorithms when applied to $S1 = \vdash 5,1,9 \vdash$.

- To check if a stack is empty :

  Require: S `-- if S1`
  1: **return** $isEmpty(S)$ `-- False`

- To get an element from a stack :

  Require: S `-- if S1`
  1: $elt \leftarrow get(S)$ `-- elt=9`

- To add an element to a stack :

  Require: S `-- if S1`
  1: $add(8, S)$ `-- S1=` $\vdash$ `5,1,9,8` $\vdash$

- To remove an element of a stack :

  Require: S `-- if S1`
  1: $remove(S)$ `-- S1=` $\vdash$ `5,1,9` $\vdash$

Stacks - Pay Attention

LIFO

Stacks are LIFO = Last In First Out (do not mix with queues, that are FIFO)

The empty stack S_0

S_0 is the empty stack. It is a notation we use. When writing “S $\leftarrow$ S_0”, it means that S is a stack (since S_0 is) and that S is now empty.

$S1 \leftarrow S_0$
$add(5, S1)$
$S_0 \leftarrow S1$

This algorithm will return an error : you can't assign a (non empty) value to a fix empty structure.

$Q \leftarrow S_0$
$L \leftarrow S_0$
$add(5, Q)$
$L \leftarrow Q$

In this algorithm, Q and L are both stacks because S_0 is a stack.

Stacks - Pay Attention

No iteration

You can't iterate on a stack. It is **NOT AN ITERABLE STRUCTURE**.

Require: S

$elt \leftarrow S[5]$

This algorithm returns an error ! A stack is not iterable.

Require: S

for elt *in* S **do**

    ...

end for

This algorithm returns an error ! A stack is not iterable.

Stacks - Pay Attention

LIFO

Stacks are LIFO = Last In First Out (do not mix with queues, that are FIFO)

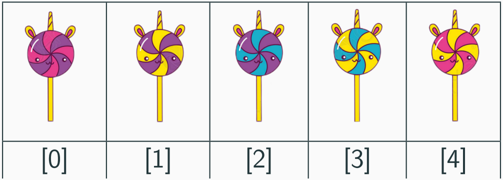
The empty stack S_0

S_0 is the empty stack. It is a notation we use. When writing "$S \leftarrow S_0$", it means that S is a stack (since S_0 is) and that S is now empty.

No iteration

You can't iterate on a stack. It is **NOT AN ITERABLE STRUCTURE**.

"I want to sort my lollipops!"



https ://www.crushpixel.com/stock-vector/vector-drawing-variety-candies-on-5907417.html

List

A List is an linear indexed structure containing some objects, organised one after the other. To each element of the list is associated an index.

Lists - Recap (2)

Operations on lists

- **add** an element at the end of the list;
- **get** the element of the list corresponding to a given index;
- **modify** the element of the list corresponding to a given index;
- **check** if the list is empty;
- a list has a length.

Notation : [ ] is the empty list.

The comments give the results of the algorithms when applied to L1 = [5,1,9].

- To check if a list is empty :

  Require: L `-- if L1`
  1: **return** $isEmpty(L)$ `-- False`

- To get an element from a list :

  Require: L `-- if L1`
  1: $elt \leftarrow L[2]$ `-- elt=9`

- To add an element to a list :

  Require: L `-- if L1`
  1: $add(8, L)$ `-- L1=[5,1,9,8]`
  2: $L1 \leftarrow L1 + [6]$ `-- [5,1,9,8,6]`

- To modify an element of a list :

  Require: L `-- if L1`
  1: $L[1] \leftarrow 3$ `-- L1= [5,3,9,8,6]`

Lists - Pay Attention

The empty list [ ]

[ ] is the empty list. It is a notation we use. When writing "L $\leftarrow$ [ ]", it means that L is a list (since [ ] is) and that L is now empty.

Iteration

You can iterate on a list. You can either use its indexes, or its elements.
Warning : when using its elements, you do not have access to its indexes anymore.

Lists - Pay Attention

Iteration

You can iterate on a list. You can either use its indexes, or its elements. Warning : when using its elements, you do not have access to its indexes anymore.

Require: L

```
for i in range(0, len(L) − 1) do
    print(L[i], i)
end for
```

This algorithm displays all the elements of the input list with their indexes.

Require: L

```
for elt in L do
    print(elt)
end for
```

This algorithm displays all the elements of the list. We do not have access to their indexes with this kind of iteration.

Lists - Pay Attention

The empty list [ ]

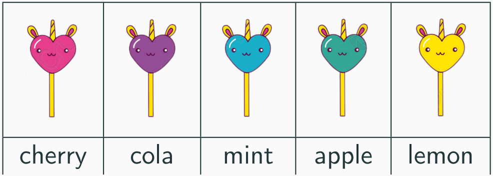
[ ] is the empty list. It is a notation we use. When writing “L $\leftarrow$ [ ]”, it means that L is a list (since [ ] is) and that L is now empty.

Iteration

You can iterate on a list. You can either use its indexes, or its elements. Warning : when using its elements, you do not have access to its indexes anymore.

Deletion

You can't delete an element from a list ! You can modify it, set its value to *None*, but not remove it. “remove(L[4])” is not a legit operation.



https ://www.crushpixel.com/stock-vector/vector-drawing-variety-candies-on-5907417.html

"I want to sort my lollipops !"

Dictionary or Associative Table

A dictionary or an associative table is a structure containing (key, value) pairs. To each value of the dictionary is associated a key. The key must be unique, while the value may be anything.

Dictionaries - Recap (2)

Operations on a dictionary

- **add** an element to the dictionary ;
- **get** the element of the dictionary corresponding to a given key ;
- **modify** the element of the dictionary corresponding to a given key ;
- **check** if the dictionary is empty ;
- a dictionary has a number of pairs : its length.

Notation : { } is the empty dictionary.

Example : D1 = {('unicorns',0), ('dragons',1)}.

- To check if a dictionary is empty :

  Require: D `-- if D1`

  1: **return** $isEmpty(D)$ `-- False`

- To get an element from a dictionary :

  Require: D `-- if D1`

  1: $elt \leftarrow D[\text{'}unicorns']$ `-- elt=0`

- To add/modify an element to a dictionary :

  Require: D `-- if D1`

  1: $D[\text{'}griffin'] \leftarrow 5$

  add pair ('griffin', 5) to D1 if 'griffin' is not in D1.keys, else modify D1['griffin'] to 5

- To check if a key is in a dictionary :

  Require: D `-- if D1`

  1: **return** $('cats'\ in\ D.keys)$ `-- False`

Dictionaries - Pay Attention

The empty dictionary { }

{ } is the empty dictionary. It is a notation we use. When writing "D $\leftarrow$ { }", it means that D is a dictionary (since { } is) and that D is now empty.

Iteration

You can iterate on a dictionary using its keys. A dictionary does not have any indexes but keys !

Dictionaries - Pay Attention

Iteration

You can iterate on a dictionary using its keys. A dictionary does not have any indexes but keys !

Require: D

for k *in* $D.keys$ **do**
    $print(D[k], k)$
end for

This algorithm displays all the elements with their keys of the input dictionary.

Dictionaries - Pay Attention

The empty dictionary $\{\ \}$

$\{\ \}$ is the empty dictionary. It is a notation we use. When writing "D $\leftarrow$ $\{\ \}$", it means that D is a dictionary (since $\{\ \}$ is) and that D is now empty.

Iteration

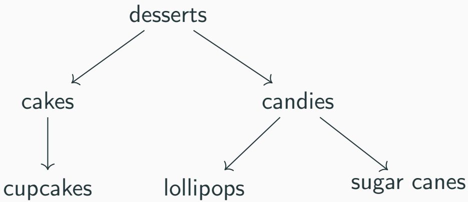
You can iterate on a dictionary using its keys. A dictionary does not have any indexes but keys !

Deletion

You can't delete an element from a dictionary ! You can modify it, set one of its value value to *None*, but not remove it. "remove(D['cat'])" is not a legit operation.

Trees - Recap (1)

"I want to organise my desserts !"



Tree

A Tree is a data structure composed of nodes and branches.

- The unique top node is the **root** ;
- Each node may have **children** (or descendants) nodes. If each node of a tree has 0 to 2 children, the tree is said binary ;
- A node that has no child (or descendant) is called a **leaf** ;
- The depth of a node is the number of its ancestors (depth of the root is 0) ;
- The height of a tree is the depth of the deepest leaf.

Operations

- **Check** if a tree is empty ;
- **Get** the **root** of a tree ;
- **Get** the **left** sub-**tree** of a tree ;
- **Get** the **right** sub-**tree** of a tree ;

We will consider only binary trees in the algorithms. We represent each tree or sub-tree by a dictionary that has 3 pairs : $\{('root', value), ('left\ child', left\ sub-tree), ('right\ child', right\ sub-tree)\}$

Example : T1 $=$
$$\begin{tikzcd}[column sep=small] & \text{'H'} \arrow[dl] \arrow[dr] & & \\ \text{'K'} \arrow[d] & & \text{'M'} \arrow[dl] \arrow[dr] & \\ \text{'P'} & \text{'D'} & & \text{'U'} \end{tikzcd}$$

- To check if a tree is empty :

  Require: T `-- if T1`

  return $isEmpty(T)$ `-- False`

- To get the root of a tree :

  Require: T `-- if T1`

  $r \leftarrow T['root']$ `-- 'H'`

- To get the left sub-tree of a tree :

  Require: T `-- if T1`

  $l \leftarrow T['left\ child']$ `-- tree 'K'→'P'`

- To get right sub-tree's root of a tree :

  Require: T `-- if T1`

  $r \leftarrow T['right\ child']['root']$ `-- 'M'`

Trees - Pay Attention

The empty tree $\{\ \}$

$\{\ \}$ is the empty tree. Since we represent trees as dictionaries, the empty dictionary is used for the empty tree. When writing "T $\leftarrow \{\ \}$", it actually means that T is a dictionary (since $\{\ \}$ is) and that T is now empty.

Single child and Leaves

If a node has a single child, it is ITS **LEFT** CHILD by default, the right one is then empty. A leaf has no children, we represent it with a node having two empty children.

Single child and Leaves

If a node has a single child, it is ITS **LEFT** CHILD by default, the right one is then empty. A leaf has no children, we represent it with a node having two empty children.

- $\{('root', N), ('left\ child', C), ('right\ child', \{\})\}$: In this tree (that may be part of a bigger tree of course), the node N has a single child C ;
- $\{('root', L), ('left\ child', \{\}), ('right\ child', \{\})\}$: In this tree (that may be part of a bigger tree of course), the node L has no children, it is a leaf.

Trees - Pay Attention

The empty tree { }

{ } is the empty tree. Since we represent trees as dictionaries, the empty dictionary is used for the empty tree. When writing "T $\leftarrow$ { }", it actually means that T is a dictionary (since { } is) and that T is now empty.

Single child and Leaves

If a node has a single child, it is ITS **LEFT** CHILD by default, the right one is then empty. A leaf has no children, we represent it with a node having two empty children.

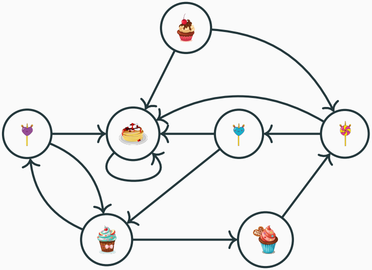
Iteration

You can't iterate on trees. But you can use a traversal procedure as we did in the labs with a queue or a stack for instance.

Deletion

You can't delete an element from a dictionary ! Hence you can't delete an element from a tree as well.

Graphs - Recap (1)



"I want to find all the desserts !"

Graph

A Graph is a data structure composed of nodes and edges.

- The nodes can be connected by edges ;
- The edges are either ALL directed or ALL un-directed :
  - If they are un-directed, both direction work ;
  - If they are directed, only the indicated direction(s) work ;
- A node can be connected to itself.

Graphs - Recap (2)

Properties

- **Size** of a graph : its number of nodes ;
- Each node has a set of **directly accessible** nodes : the nodes linked to them by an edge ;
- We say that a node B is **accessible** from a node A if there exist a **path** from node A to node B ;
- **Length of a path** between two nodes : the number of edges used to go from one to the other.

Operations on graphs

Note that if the graph is directed the direction of the edges must be specified.

- **Creating an edge** between two existing nodes.
  → *Create the edge* $1 \rightarrow 3$.
- **Creating a node**.
  → *Create the node 8.*
- **Removing an edge**.
  → *Remove the edge* $5 \rightarrow 4$.
- **Removing a node** : removing an existing node from a graph and all the edges coming from or to this node.
  → *Remove the node* 3.

Graphs - Pay Attention

Adding edges

When adding an edge, you need both nodes (starting and ending node of the edge) to exist. You can't add an edge if one of its nodes does not already exist in the graph.

Deletion

You can delete an element from a graph. When removing an edge, nothing more happens, but when removing a node, all the edges linked to this node are also removed.

Some legit "one-step" instructions :

Add :

- the node X ;
- the edge Z $\rightarrow$ Y (provided BOTH nodes Z and Y EXIST in the graph.

Delete :

- the node X ;
- the edge X $\rightarrow$ Y ;
- all the edges coming to node X ;
- all the edges coming from node X.

Informal iteration

You can use two methods to go through a graph : exploring the paths starting from a node of your choice, or "for each node ...".
Warning : when exploring all the paths, pay attention not to forget (or delete) a possible path, or some elements you wanted to explore.

If you choose "**for each node of the graph**, ..." :

- there is no (defined) starting node ;
- you do not have a specific order to explore the graph ;
- if you delete some edges/nodes, it will not have a consequence on what you are doing.

If you choose to **explore the paths** :

- you need a starting node ;
- the way you explore the graph is ordered (neighbour after neighbour) ;
- if you modify the graph (like deleting some edges/nodes) on you way, it may have an impact on what you are doing (inaccessible nodes, etc.).

Informal iteration

You can use two methods to go through a graph : exploring the paths starting from a node of your choice, or "for each node ...".
Warning : when exploring all the paths, pay attention not to forget (or delete) a possible path, or some elements you wanted to explore.

Note : When you want some information, if you can have it in one step (i.e., the node, its direct neighbours, from which nodes it is direct neighbours), then you can write it just like that. Else (you need mode than one step), you need to do something (exploring the paths, going through the graph, etc.).

Graphs - Pay Attention

Adding edges

When adding an edge, you need both nodes (starting and ending node of the edge) to exist. You can't add an edge if one of its nodes does not already exist in the graph.

Deletion

You can delete an element from a graph. When removing an edge, nothing more happens, but when removing a node, all the edges linked to this node are also removed.

Informal iteration

You can use two methods to go through a graph : exploring the paths starting from a node of your choice, or "for each node ...".
Warning : when exploring all the paths, pay attention not to forget (or delete) a possible path, or some elements you wanted to explore.

About queues and stack use

You can add whatever you want to a queue or a stack. When adding (or removing) something (an integer, a list, a tree, a graph, etc.) to a queue or a stack, you add the whole object. You do not need to cut it in parts and add parts after parts. (The same goes for setting values to elements of the lists.)
$\rightarrow$ *If you add a person to a CROUS queue, you add the whole person. You do not behead them, cut their arms and legs, and add first the head, then the arms, then the body, then the legs. Same idea when someone finally have their lunch and quit the queue !*

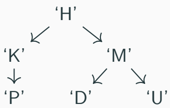
About queues and stack use

You can add whatever you want to a queue or a stack. When adding (or removing) something (an integer, a list, a tree, a graph, etc.) to a queue or a stack, you add the whole object. You do not need to cut it in parts and add parts after parts. (The same goes for setting values to elements of the lists.)

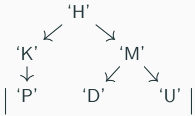
- **Require:** T `-- a tree`

  $Q \leftarrow Q_0$
  $add(T, Q)$

Example : T1=



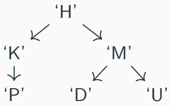
Q=



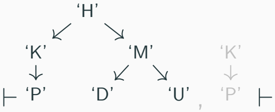
- **Require:** T `-- a tree`

  $S \leftarrow S_0$
  $add(T, S)$
  $add(T['left\ child'], S)$
  $remove(S)$

Example : T1=



S=



Exercise

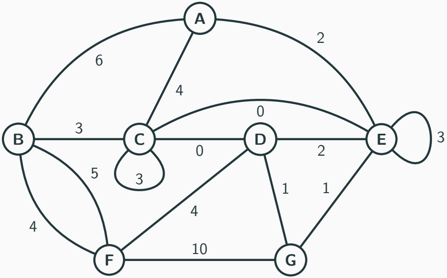
Dijkstra Algorithm (1)

Dijkstra Algorithm

Idea : Find the path of minimal weight from a node to another one in a weighted graph.

Weighted Graph

A weighted graph is a graph where each edge has a (numerical) value, called a weight. When talking about the “shortest path” in a weighted graph, we usually mean the “lighter path”, *i.e.*, the one of minimal weight.



Dijkstra Algorithm (2)

Dijkstra Algorithm

- **Input** : a weighted graph, which weights are positive integers, and a starting node X ;
- *Initialisation + Main part (see next slides) ;*
- **Output** : a table allowing to know the minimal path weight from the chosen starting node to each of the other nodes. This table also allow to find the minimal path associated to the minimal path weight.

Dijkstra Algorithm (2)

Dijkstra Algorithm's Initialisation (first step)

- Draw a table having one line per node. The columns will be the steps of the algo. In the starting node's line, write 0 in the first column (i.e., first step) and in all other nodes line, write ∞ in the first column ;
- *Main part (see next slide) ;*

	step1	step2	step3	step4	step5	step6	step7	step8
A								
B								
C								
D								
E								
F								
G								

Dijkstra Algorithm (2)

Dijkstra Algorithm's Initialisation (first step)

- Draw a table having one line per node. The columns will be the steps of the algo. In the starting node's line, write 0 in the first column (i.e., first step) and in all other nodes line, write ∞ in the first column ;
- *Main part (see next slide) ;*

	step1	step2	step3	step4	step5	step6	step7	step8
A	0							
B	∞							
C	∞							
D	∞							
E	∞							
F	∞							
G	∞							

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [detailed]

For each following step :

- in the previous column (*i.e.*, previous algo step) chose one minimal value (one path of minimal weight) and underline it ;
- it correspond to a node Y (the node corresponding to the line) ;
- write / in all following columns for this node : you found the minimal path to access it from X ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them ;
- if the computed path is smaller than the one you had before, write its new (smaller) value in the current column of the table ;
- repeat until all the minimal paths have been chosen/found ;

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	0							
B	∞							
C	∞							
D	∞							
E	∞							
F	∞							
G	∞							

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	0							
B	∞							
C	∞							
D	∞							
E	∞							
F	∞							
G	∞							

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	**0**	/	/	/	/	/	/	/
B	∞	6						
C	∞	4						
D	∞	∞						
E	∞	2						
F	∞	∞						
G	∞	∞						

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	$\underline{\mathbf{0}}$	/	/	/	/	/	/	/
B	∞	6						
C	∞	4						
D	∞	∞						
E	∞	2						
F	∞	∞						
G	∞	∞						

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	**0**	/	/	/	/	/	/	/
B	∞	6	6					
C	∞	4	4					
D	∞	∞	4					
E	∞	**2**	/	/	/	/	/	/
F	∞	∞	∞					
G	∞	∞	3					

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	<u>**0**</u>	/	/	/	/	/	/	/
B	∞	6	6					
C	∞	4	4					
D	∞	∞	4					
E	∞	<u>**2**</u>	/	/	/	/	/	/
F	∞	∞	∞					
G	∞	∞	3					

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	**0**	/	/	/	/	/	/	/
B	∞	6	6	6				
C	∞	4	4	4				
D	∞	∞	4	4				
E	∞	**2**	/	/	/	/	/	/
F	∞	∞	∞	13				
G	∞	∞	**3**	/	/	/	/	/

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	**0**	/	/	/	/	/	/	/
B	∞	6	6	6				
C	∞	4	4	4				
D	∞	∞	4	4				
E	∞	**2**	/	/	/	/	/	/
F	∞	∞	∞	13				
G	∞	∞	**3**	/	/	/	/	/

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	$\underline{\mathbf{0}}$	/	/	/	/	/	/	/
B	∞	6	6	6	6			
C	∞	4	4	$\underline{\mathbf{4}}$	/	/	/	/
D	∞	∞	4	4	4			
E	∞	$\underline{\mathbf{2}}$	/	/	/	/	/	/
F	∞	∞	∞	13	13			
G	∞	∞	$\underline{\mathbf{3}}$	/	/	/	/	/

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	**0**	/	/	/	/	/	/	/
B	∞	6	6	6	6			
C	∞	4	4	**4**	/	/	/	/
D	∞	∞	4	4	4			
E	∞	**2**	/	/	/	/	/	/
F	∞	∞	∞	13	13			
G	∞	∞	**3**	/	/	/	/	/

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	**0**	/	/	/	/	/	/	/
B	∞	6	6	6	6	6		
C	∞	4	4	**4**	/	/	/	/
D	∞	∞	4	4	**4**	/	/	/
E	∞	**2**	/	/	/	/	/	/
F	∞	∞	∞	13	13	8		
G	∞	∞	**3**	/	/	/	/	/

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	**0**	/	/	/	/	/	/	/
B	∞	6	6	6	6	6		
C	∞	4	4	**4**	/	/	/	/
D	∞	∞	4	4	**4**	/	/	/
E	∞	**2**	/	/	/	/	/	/
F	∞	∞	∞	13	13	8		
G	∞	∞	**3**	/	/	/	/	/

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	**0**	/	/	/	/	/	/	/
B	∞	6	6	6	6	**6**	/	/
C	∞	4	4	**4**	/	/	/	/
D	∞	∞	4	4	**4**	/	/	/
E	∞	**2**	/	/	/	/	/	/
F	∞	∞	∞	13	13	8	8	
G	∞	∞	**3**	/	/	/	/	/

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	**0**	/	/	/	/	/	/	/
B	∞	6	6	6	6	**6**	/	/
C	∞	4	4	**4**	/	/	/	/
D	∞	∞	4	4	**4**	/	/	/
E	∞	**2**	/	/	/	/	/	/
F	∞	∞	∞	13	13	8	8	
G	∞	∞	**3**	/	/	/	/	/

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	**0**	/	/	/	/	/	/	/
B	∞	6	6	6	6	**6**	/	/
C	∞	4	4	**4**	/	/	/	/
D	∞	∞	4	4	**4**	/	/	/
E	∞	**2**	/	/	/	/	/	/
F	∞	∞	∞	13	13	8	**8**	/
G	∞	∞	**3**	/	/	/	/	/

Dijkstra Algorithm (2)

Dijkstra Algorithm's Main part [summarized]

For each step (repeat until all the minimal paths have been found) :

- Chose one minimal value in the previous column and write / in all following columns for this node Y ;
- for all neighbours N of Y, compute the total path weight if you go from (X to) Y to them, if the computed path is smaller than the one you had before, change its value in the table ;

	step1	step2	step3	step4	step5	step6	step7	step8
A	**0**	/	/	/	/	/	/	/
B	∞	6	6	6	6	**6**	/	/
C	∞	4	4	**4**	/	/	/	/
D	∞	∞	4	4	**4**	/	/	/
E	∞	**2**	/	/	/	/	/	/
F	∞	∞	∞	13	13	8	**8**	/
G	∞	∞	**3**	/	/	/	/	/