Data Structures (Not UML)

Pseudo-code Syntax, Lists, Dictionaries

Amandine Decker & Marie Cousin

M1 TAL/SC 2023–2024

Université de Lorraine, LORIA

Class Organisation

First Lab :

- Reminder : optional exercise (due October 16th) ;
- hand back on a sheet of paper (October 13th), send it by e-mail, or hand back on Arche.
- Correction of the first lab (except bonus exercise) will be uploaded today (October 6th).

How to find the Arche repository :

- on your Arche top bar (blue-ish), click on “Home” (“Accueil”) ;
- then “LORRAINE MANAGEMENT” (purple) ;
- and search for “Data Structure” ;
- Full name of the course is “Data Structures - Beginners”, password is “Nox”.

Syntax

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.

- Create and initialise a variable : $new\_var \leftarrow init\_value$
- Modify a variable : $var \leftarrow new\_value$
  - Create and initialise empty structures like last week :

    1: $S \leftarrow S_0$ -- *creates an empty stack*
  - But it can work with any type of data :

    1: $c \leftarrow 5$ -- *creates a variable c of value 5*

    2: $c \leftarrow 0$ -- *modifies the variable c, it is now of value 0*

    3: $c \leftarrow c + 3$ -- *modifies the variable c, it is now of value 3 (0 + 3)*

Booleans and Negation

Boolean

A boolean is a type of object that can have the value *true* or *false*. These values (*true* and *false*) can be assigned to a variable.

- Assign true or false to a variable : $var \leftarrow true$; var is a boolean.
- Booleans are used in **if** and **while** loops.

Negation

The negation symbol is !, and it is used to change the value of a boolean.

- $!true = false$
- $!false = true$

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.

```
if condition then
    instruction 1
    instruction 2
    etc.
else    -- optional part
    instruction A
    instruction B
    etc.
end if    -- but this one is mandatory
```

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. The instructions 1, 2, etc. will be executed as many times as there are elements.

Note : The loop variable (here X) can be used in the loop.

```
for X in elements do
    instruction 1
    instruction 2
    etc.
end for
```

Loops (2)

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).

```
while loop condition do
    instruction 1
    instruction 2
    etc.
end while
```

Return VS Print

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 !****)*

1: **Return** *result*

Print instruction

This instruction will display *data* and the execution of the algorithm will continue after it. If you want an algorithm to output something, you should use **return** and not **print**.

1: **Print** *data* -- *of whatever type*

Some Tipps

- If I create an object or a variable : I initialise it ;

  1: *new_variable* $\leftarrow$ *init_value*

- If I want to count things : I create a variable, that will be my counter, and I modify it to count things ;

  1: *counter* $\leftarrow 0$
  2: *counter* $\leftarrow$ *counter* $+ 1$ -- *if you count one thing at a time*

- If I want to remove (/delete/overwrite/...) some data I may need later : I create a variable to store that data.

  1: -- *suppose you have a queue Q and you want to access the second element but you will need the first one later*
  2: *current* $\leftarrow$ *get*(Q)
  3: *remove*(Q)

Most Important Tip

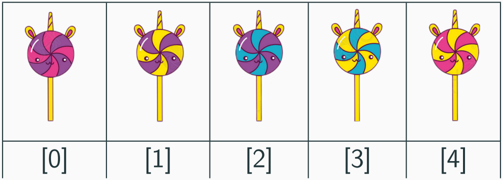
Remain consistent and logical! If you do not write your algorithm exactly like in the course but use a formalism that is understandable and remains the same throughout your work it will be alright. Examples :

- Do not use $\leftarrow$ on one line and $=$ like in Python in the next one;
- Do not use $\rightarrow$ instead of $\leftarrow$ because it does not make sense in terms of what is really happening (in $var \leftarrow e$, var becomes e, it receives the value e, etc.).

If you have any doubt on the way you write something, ask us! Or describe precisely what you mean with a certain notation. Always comment and explain what you do.

Lists

Lists : Idea



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

"I want to sort my lollipops !"

- Nox wants to sort his lollipops ;
- In a basic shelf, he puts one lollipop per compartment, one after the other ;
- We have a list of lollipops !

Lists : Definition

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 : Definition

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.

$\rightarrow$ *Like the lists you are used to in human life : grocery list, todo list, etc.*

Operations

- 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, we can get its number of elements.

Some Formalism

If we consider a set of elements E, we can write $Lists(E)$ the set of all the lists containing elements of E. The empty list $[\,]$ is in $Lists(E)$.

If $e \in E$, and $L, L' \in Lists(E)$, the following operations exist :

- $len(L)$: returns the length of L ;
- $L[i] \leftarrow e$: sets the value of $L[i]$ to e provided that $i < len(L)$;
- $L[i]$: returns the value e of $L[i]$ provided that $i < len(L)$;
- $L[i : j]$: returns a list L' corresponding to the sub-list between indexes i and $j-1$ of L provided that $i < len(L)$ and $j \leq len(L)$;
- $add(e, L)$: adds the element e at the end of L ;
- $L + L'$ returns the concatenation of L and L' (*i.e.*, the elements of L followed by the elements of L') ;

and the following properties :

- $isEmpty([\,]) = \text{True}$
- $add(e, L) = L :: e$
- $isEmpty(add(e, L)) = \text{False}$
- if $len(L) = n$ then $len(add(e, L)) = n + 1$

Lists : How to Write it ?

- Create the empty list :

```
L ← [ ]
```

- Given an already existing list L, we can add an element, for instance 5 :

```
add(5, L)
```

- Modify the 6th element of a given list :

```
idx ← 6
if idx < len(L) then
    L[idx] ← 85
end if
```

- Do something with all the elements of a list one after the other :

```
L ← [2, 4, 1, 8, 4]
for x in L do
    print(x)      -- Or anything you would like to do with x
end for
```

Lists : Example

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

2	1	5	7	3

Lists : Example

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

Lists : Example

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

Lists : Example

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

Lists : Example

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

Lists : Example

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

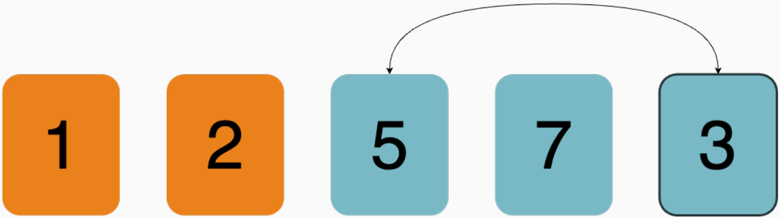
Lists : Example

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.



Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

Lists : Example

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

Lists : Example

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

Lists : Example

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

Lists : Example

Sort

Given an unordered list of natural integers, we sort them from the smallest to the largest one, by finding the smallest element of the list, exchanging it with the first one, and repeating the process but at the next place. The next smallest element will go at the second place, etc.

Lists : Example

```
Require: L
for i in [0, len(L) − 1] do    -- i: index of the future smallest element
    min ← i    -- index of the smallest not-yet-sorted element of L
    for j in [i, len(L)] do    -- to find the smallest element
        if L[j] < L[min] then
            min ← j
        end if
    end for
    buffer ← L[i]
    L[i] ← L[min]
    L[min] ← buffer
end for
return L
```

Dictionaries / Associative Tables

Dictionaries / Associative Tables : Idea

"I want to sort my lollipops !"

				
cherry	cola	mint	apple	lemon

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

- Nox wants to sort his lollipops by flavor ;
- In a shelf, he puts one lollipop per compartment. The compartments are labelled with a chosen key, the flavor ;
- We have a dictionary of lollipops !

Dictionaries / Associative Tables : Definition

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 / Associative Tables : Definition

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.

→ *Like the dictionary you are used to in human life : a word dictionary, etc.*

Operations

- 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, we can get it.

Some Formalism

If we consider a set of elements E, and a set of keys K we can write $Dict(K, E)$ the set of all the tables (or dictionaries) containing pairs of keys of K and elements of E. The empty table/dictionary $\{\}$ is in $Dict(K, E)$.

If $e \in E$, $k \in K$ and $D \in Dict(K, E)$, the following operations exist :

- $isEmpty(D)$: returns *true* or *false* depending on whether D is empty or not ;
- $len(D)$: returns the number of (key, value)-pairs of D ;
- $D.keys()$: returns the keys of D ;
- $D[k] \leftarrow e$: sets the value of $D[k]$ to e provided that $k \in D.keys$;
- $D[k]$: returns the value e of $D[k]$ provided that that $k \in D.keys$;
- $add(k, e, D)$ (or based on the Python syntax : $D[k] \leftarrow e$) : adds the couple (k, e) to D provided that that $k \notin D.keys$.

Dictionaries / Associative Tables : How to write

- Create the empty dictionary :

  $D \leftarrow \{\}$

- Given an already existing dictionary, we can add a (key, value)-pair, for example $(raspberry, 3)$:

  if $rasperry \notin D.keys$ **then**
  $\quad add(raspberry, 3, D)$ `-- or` $D[k] \leftarrow 3$
  end if

Dictionaries / Associative Tables : How to write

- Modify the value of a key (for instance *raspberry*) of a given dictionary :

```
if rasperry ∈ D.keys then
    D[raspberry] ← 8
end if
```

- Do something with all the pairs of a dictionary one after the other :

```
D ← {(blue, elephant), (red, bird), (orange, tiger), (yellow, lion)}
for x in D.keys do
    print(D[x])      -- Or anything you would like to do with x
end for
```

Dictionaries / Associative Tables : Example

Count occurrences

Given a list of words, we want to know how many times each word appears.

Dictionaries / Associative Tables : Example

Count occurrences

Given a list of words, we want to know how many times each word appears.

["cherry", "mint", "apple", "mint", "cola", "mint", "apple", "lemon", "mint", "cola", "cola"]

Dictionaries / Associative Tables : Example

Count occurrences

Given a list of words, we want to know how many times each word appears.

["cherry", "mint", "apple", "mint", "cola", "mint", "apple", "lemon", "mint", "cola", "cola"]

"cherry" : 1

Dictionaries / Associative Tables : Example

Count occurrences

Given a list of words, we want to know how many times each word appears.

["cherry", "mint", "apple", "mint", "cola", "mint", "apple", "lemon", "mint", "cola", "cola"]

"cherry" : 1

"mint" : 4

Dictionaries / Associative Tables : Example

Count occurrences

Given a list of words, we want to know how many times each word appears.

["cherry", "mint", "apple", "mint", "cola", "mint", "apple", "lemon", "mint", "cola", "cola"]

"cherry" : 1

"mint" : 4

"apple" : 2

Dictionaries / Associative Tables : Example

Count occurrences

Given a list of words, we want to know how many times each word appears.

["cherry", "mint", "apple", "mint", "cola", "mint", "apple", "lemon", "mint", "cola", "cola"]

"cherry" : 1

"mint" : 4

"apple" : 2

"cola" : 3

Dictionaries / Associative Tables : Example

Count occurrences

Given a list of words, we want to know how many times each word appears.

["cherry", "mint", "apple", "mint", "cola", "mint", "apple", "lemon", "mint", "cola", "cola"]

"cherry" : 1

"mint" : 4

"apple" : 2

"cola" : 3

"lemon" : 1

Dictionaries / Associative Tables : Example

Require: L

$D \leftarrow \{\}$
for x *in* L **do**
    if x in D.keys **then**
        $D[x] \leftarrow D[x] + 1$ `-- Add 1 to the right counter`
    else
        $D[x] \leftarrow 1$ `-- Create a new key and initialise the counter to 1`
    end if
end for
return D

Summary

Summary

- Lists are a data structure that has an index ;
- The elements in a list are accessible via this index ;
- Dictionaries or associative tables are another data structure, containing (key, value)-pairs ;
- The elements in a dictionary are accessible via the keys ;
- Before doing anything in a dictionary, you have to check if the key exists.