Structures linéaires

Informatique pour tou(te)s - semaines 37 à 40

Guillaume CONNAN

septembre 2015

Lycée Clemenceau - MP / MP*

Hiérarchie de la mémoire

Tableau (array, vector) statique

Tableau (array, vector) statique

Liste chaînée dynamique pointeur

Liste chaînée dynamique pointeur

Liste chaînée dynamique pointeur

Liste Python...un peu des deux.

Liste Python...un peu des deux.

Python...c'est du C !

Python...c'est du C !

```c
typedef struct {
        PyObject_VAR_HEAD
        /* Vector of pointers to list elements.  list[0] is ob_item[0], etc. */
        PyObject **ob_item;

        /* ob_item contains space for 'allocated' elements.  The number
         * currently in use is ob_size.
         * Invariants:
         *     0 <= ob_size <= allocated
         *     len(list) == ob_size
         *     ob_item == NULL implies ob_size == allocated == 0
         * list.sort() temporarily sets allocated to -1 to detect mutations.
         *
         * Items must normally not be NULL, except during construction when
         * the list is not yet visible outside the function that builds it.
         */
        Py_ssize_t allocated;
    } PyListObject;
```

Obtenir un élément

```
static PyObject *
list_item(PyListObject *a, int i)
{
if (i < 0 || i >= a->ob_size) {
if (indexerr == NULL)
indexerr = PyString_FromString(
"list index out of range");
PyErr_SetObject(PyExc_IndexError, indexerr);
return NULL;
}
Py_INCREF(a->ob_item[i]);
return a->ob_item[i];
}
```

Test d'appartenance

```
static int
list_contains(PyListObject *a, PyObject *el)
{
int i;

for (i = 0; i < a->ob_size; ++i) {
int cmp = PyObject_RichCompareBool(el, PyList_GET_ITEM(a, i),
   Py_EQ);
if (cmp > 0)
return 1;
else if (cmp < 0)
return -1;
}
return 0;
}
```

Linguiste ≠ locuteur natif

Objectif ≠ connaître Python

Linguiste ≠ locuteur natif
Objectif ≠ connaître Python

Abstraction

Bug de l'an 2000

Préparer celui de l'an 10 000

Abstraction

Bug de l'an 2000

Préparer celui de l'an 10 000

Abstraction
Bug de l'an 2000
Préparer celui de l'an 10 000

Extrait de l'épreuve X-ENS (MP option SI et PC) 2015

Dans la suite nous aurons besoin d'utiliser des piles d'entiers, dont on rappelle la définition ci-dessous :

Définition 2 *Une pile d'entiers est une structure de données permettant de stocker des entiers et d'effectuer les opérations suivantes en temps constant (indépendant de la taille de la pile) :*

- *créer une nouvelle pile vide,*
- *déterminer si la pile est vide,*
- *insérer un entier au sommet de la pile,*
- *déterminer la valeur de l'entier au sommet de la pile,*
- *retirer l'entier au sommet de la pile.*

Nous supposerons fournies les fonctions suivantes, qui réalisent les opérations ci-dessus et s'executent chacune en temps constant :

- newStack(), qui ne prend pas d'argument et renvoie une pile vide,
- isEmpty(s), qui prend une pile s en argument et renvoie True ou False suivant que s est vide ou non,
- push(i, s), qui prend un entier i et une pile s en argument, insère i au sommet de s (c'est-à-dire à la fin de la liste), et ne renvoie rien,
- top(s), qui prend une pile s (supposée non vide) en argument et renvoie la valeur de l'entier au sommet de s (c'est-à-dire à la fin de la liste),
- pop(s), qui prend une pile s (supposée non vide) en argument, supprime l'entier au sommet de s (c'est-à-dire à la fin de la liste) et renvoie sa valeur.

Dans la suite il est demandé aux candidats de manipuler les piles uniquement au travers de ces fonctions, sans aucune hypothèse sur la représentation effective des piles en mémoire.

- Séparation représentation / utilisation
- Un type : ensemble de valeurs et ensemble d'opérations pour créer, modifier, manipuler ces valeurs (monoïde, ...)

- Séparation représentation / utilisation
- Un type : ensemble de valeurs et ensemble d'opérations pour créer, modifier, manipuler ces valeurs (magma, monoïde,...)

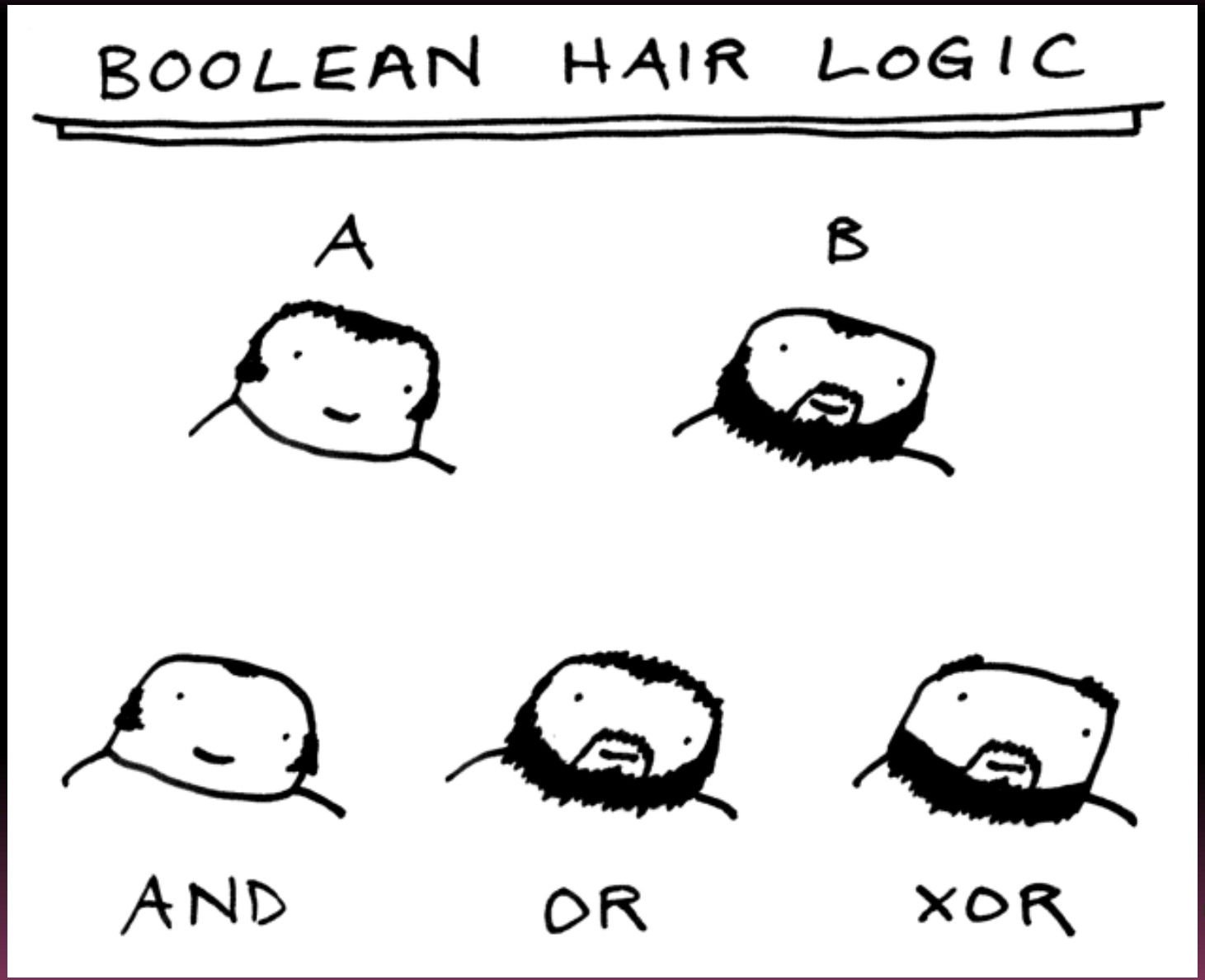
- Séparation représentation / utilisation
- Un type : ensemble de valeurs et ensemble d'opérations pour créer, modifier, manipuler ces valeurs (magma, monoïde,...)
- Barrière d'abstraction

- Séparation représentation / utilisation
- Un type : ensemble de valeurs et ensemble d'opérations pour créer, modifier, manipuler ces valeurs (magma, monoïde,...)
- Barrière d'abstraction
- Séparer la programmation de la représentation du type et la programmation des algorithmes qui utilisent ce type

- Séparation représentation / utilisation
- Un type : ensemble de valeurs et ensemble d'opérations pour créer, modifier, manipuler ces valeurs (magma, monoïde,...)
- Barrière d'abstraction
- Séparer la programmation de la représentation du type et la programmation des algorithmes qui utilisent ce type

- Séparation représentation / utilisation
- Un type : ensemble de valeurs et ensemble d'opérations pour créer, modifier, manipuler ces valeurs (magma, monoïde,...)
- Barrière d'abstraction
- Séparer la programmation de la représentation du type et la programmation des algorithmes qui utilisent ce type

Les Booléens



Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul

- Faux ∧ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul (constructeur)
  - Faux ∧ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }

- Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }
  - Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }

- Faux ∧ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }

- Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }
  - $\_ \wedge \_ : :$ Boul $\times$ Boul $\longrightarrow$ Boul
    - Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }
  - $\neg$: : Boul $\longrightarrow$ Boul
  - $\wedge$: : Boul Boul $\longrightarrow$ Boul
- **Axiomes :**
  - Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }
  - $\neg$: : Boul $\longrightarrow$ Boul
  - $\wedge$: : Boul Boul $\longrightarrow$ Boul
- **Axiomes**
  - Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }
  - $\neg$: : Boul $\longrightarrow$ Boul
  - $\wedge$: : Boul Boul $\longrightarrow$ Boul
- **Axiomes**
  - Variable : b : : Boul
  - Équations :
    - Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }
  - $\neg$: : Boul $\longrightarrow$ Boul
  - $\wedge$: : Boul Boul $\longrightarrow$ Boul
- **Axiomes**
  - **Variable :** b : : Boul
  - **Équations :**
    - Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }
  - $\neg$: : Boul $\longrightarrow$ Boul
  - $\wedge$: : Boul Boul $\longrightarrow$ Boul
- **Axiomes**
  - **Variable :** b : : Boul
  - **Équations :**
    - $\neg$ Vrai = Faux
    - $\neg$ Faux = Vrai
    - Vrai $\wedge$ b = b
    - Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }
  - $\neg$: : Boul $\longrightarrow$ Boul
  - $\wedge$: : Boul Boul $\longrightarrow$ Boul
- **Axiomes**
  - **Variable :** b : : Boul
  - **Équations :**
    - $\neg$ Vrai = Faux
    - $\neg$ Faux = Vrai
    - Vrai $\wedge$ b = b
    - Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }
  - $\neg$: : Boul $\longrightarrow$ Boul
  - $\wedge$: : Boul Boul $\longrightarrow$ Boul
- **Axiomes**
  - **Variable :** b : : Boul
  - **Équations :**
    - $\neg$ Vrai = Faux
    - $\neg$ Faux = Vrai
    - Vrai $\wedge$ b = b
    - Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }
  - $\neg$: : Boul $\longrightarrow$ Boul
  - $\wedge$: : Boul Boul $\longrightarrow$ Boul
- **Axiomes**
  - **Variable :** b : : Boul
  - **Équations :**
    - $\neg$ Vrai = Faux
    - $\neg$ Faux = Vrai
    - Vrai $\wedge$ b = b
    - Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }
  - $\neg$: : Boul $\longrightarrow$ Boul
  - $\wedge$: : Boul Boul $\longrightarrow$ Boul
- **Axiomes**
  - **Variable :** b : : Boul
  - **Équations :**
    - $\neg$ Vrai = Faux
    - $\neg$ Faux = Vrai
    - Vrai $\wedge$ b = b
    - Faux $\wedge$ b = Faux

Une spécification des booléens

- **Sorte :** Boul
- **Opérations :**
  - Vrai : : Boul { constructeur }
  - Faux : : Boul { constructeur }
  - $\neg$: : Boul $\longrightarrow$ Boul
  - $\wedge$: : Boul Boul $\longrightarrow$ Boul
- **Axiomes**
  - **Variable :** b : : Boul
  - **Équations :**
    - $\neg$ Vrai = Faux
    - $\neg$ Faux = Vrai
    - Vrai $\wedge$ b = b
    - Faux $\wedge$ b = Faux

Et les autres opérations ? ($\vee$, $\Longrightarrow$, $\Longleftrightarrow$ et $\oplus$)

- $b \vee c = \neg((\neg b) \wedge (\neg c))$
- $b \Longrightarrow c = (\neg b) \vee c$
- $b \Longleftrightarrow c = (b \Longrightarrow c) \wedge (c \Longrightarrow b)$
- $b \oplus c = (b \vee c) \wedge \neg(b \wedge c)$

Et les autres opérations ? ($\vee$, $\Longrightarrow$, $\Longleftrightarrow$ et $\oplus$)

- $b \vee c = \neg((\neg b) \wedge (\neg c))$
- $b \Longrightarrow c = (\neg b) \vee c$
- $b \Longleftrightarrow c = (b \Longrightarrow c) \wedge (c \Longrightarrow b)$
- $b \oplus c = \neg(b \Longleftrightarrow c)$

Et les autres opérations ? ($\vee$, $\Longrightarrow$, $\Longleftrightarrow$ et $\oplus$)

- $b \vee c = \neg((\neg b) \wedge (\neg c))$
- $b \Longrightarrow c = (\neg b) \vee c$
- $b \Longleftrightarrow c = (b \Longrightarrow c) \wedge (c \Longrightarrow b)$
- $b \oplus c = \neg(b \Longleftrightarrow c)$

Et les autres opérations ? ($\vee$, $\Longrightarrow$, $\Longleftrightarrow$ et $\oplus$)

- $b \vee c = \neg((\neg b) \wedge (\neg c))$
- $b \Longrightarrow c = (\neg b) \vee c$
- $b \Longleftrightarrow c = (b \Longrightarrow c) \wedge (c \Longrightarrow b)$
- $b \oplus c = \neg(b \Longleftrightarrow c)$

Et les autres opérations ? ($\vee$, $\Longrightarrow$, $\Longleftrightarrow$ et $\oplus$)

- $b \vee c = \neg((\neg b) \wedge (\neg c))$
- $b \Longrightarrow c = (\neg b) \vee c$
- $b \Longleftrightarrow c = (b \Longrightarrow c) \wedge (c \Longrightarrow b)$
- $b \oplus c = \neg(b \Longleftrightarrow c)$

Une spécification des listes chaînées

Définition 1 (Liste)

Une liste sur un ensemble E est soit le *Vide* soit un élément de E suivi d'une liste.

(Comparez avec cette définition d'un mot : c'est le mot vide ou une lettre suivie d'un mot...).

Une spécification des listes chaînées

Définition 1 (Liste)

Une liste sur un ensemble E est soit le *Vide* soit un élément de E suivi d'une liste.

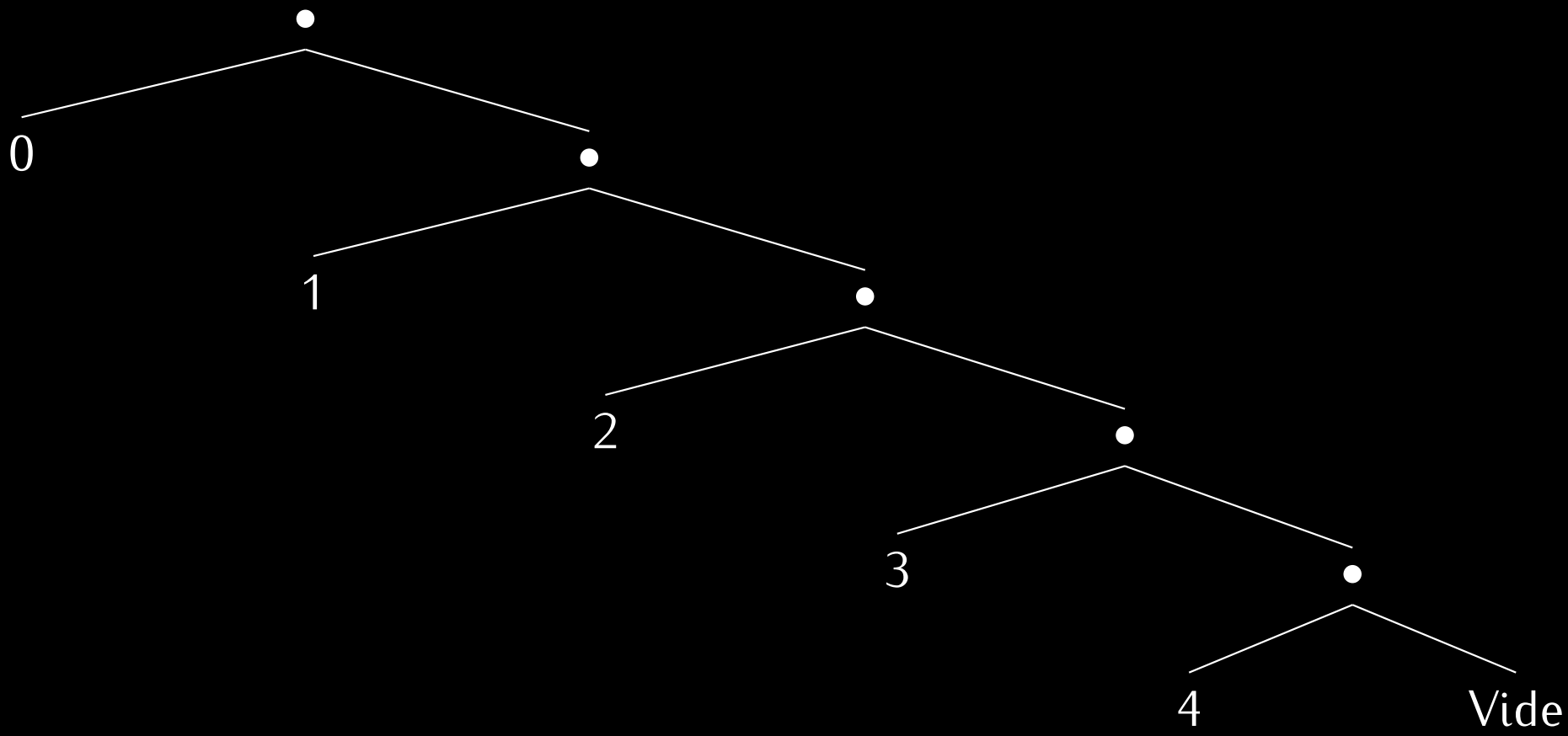
(*Comparez avec cette définition d'un mot : c'est le mot vide ou une lettre suivie d'un mot...*).

Une spécification des listes chaînées

Définition 1 (Liste)

Une liste sur un ensemble E est soit le *Vide* soit un élément de E suivi d'une liste.

(*Comparez avec cette définition d'un mot : c'est le mot vide ou une lettre suivie d'un mot...*).


0
1
2
3
4
Vide

- Il nous faut disposer d'une liste vide.
- Il faut pouvoir ajouter un élément en tête
- sélectionner la tête
- sélectionner la liste privée de la tête
- tester si une liste est vide
- on ne pourra pas demander la tête ou la queue d'une liste vide.
- une liste ayant une tête n'est pas vide
- si on a ajouté un élément à une liste, sa tête est cet élément et la queue est la liste initiale

- Il nous faut disposer d'une liste vide.
- Il faut pouvoir ajouter un élément en tête
- sélectionner la tête
- sélectionner la liste privée de la tête
- tester si une liste est vide
- on ne pourra pas demander la tête ou la queue d'une liste vide.
- une liste ayant une tête n'est pas vide
- si on a ajouté un élément à une liste, sa tête est cet élément et la queue est la liste initiale

- Il nous faut disposer d'une liste vide.
- Il faut pouvoir ajouter un élément en tête
- sélectionner la tête
- sélectionner la liste privée de la tête
- tester si une liste est vide
- on ne pourra pas demander la tête ou la queue d'une liste vide.
- une liste ayant une tête n'est pas vide
- si on a ajouté un élément à une liste, sa tête est cet élément et la queue est la liste initiale

- Il nous faut disposer d'une liste vide.
- Il faut pouvoir ajouter un élément en tête
- sélectionner la tête
- sélectionner la liste privée de la tête
- tester si une liste est vide
- on ne pourra pas demander la tête ou la queue d'une liste vide.
- une liste ayant une tête n'est pas vide
- si on a ajouté un élément à une liste, sa tête est cet élément et la queue est la liste initiale

- Il nous faut disposer d'une liste vide.
- Il faut pouvoir ajouter un élément en tête
- sélectionner la tête
- sélectionner la liste privée de la tête
- tester si une liste est vide
- on ne pourra pas demander la tête ou la queue d'une liste vide.
- une liste ayant une tête n'est pas vide
- si on a ajouté un élément à une liste, sa tête est cet élément et la queue est la liste initiale

- Il nous faut disposer d'une liste vide.
- Il faut pouvoir ajouter un élément en tête
- sélectionner la tête
- sélectionner la liste privée de la tête
- tester si une liste est vide
- on ne pourra pas demander la tête ou la queue d'une liste vide.
- une liste ayant une tête n'est pas vide
- si on a ajouté un élément à une liste, sa tête est cet élément et la queue est la liste initiale

- Il nous faut disposer d'une liste vide.
- Il faut pouvoir ajouter un élément en tête
- sélectionner la tête
- sélectionner la liste privée de la tête
- tester si une liste est vide
- on ne pourra pas demander la tête ou la queue d'une liste vide.
- une liste ayant une tête n'est pas vide
- si on a ajouté un élément à une liste, sa tête est cet élément et la queue est la liste initiale

- Il nous faut disposer d'une liste vide.
- Il faut pouvoir ajouter un élément en tête
- sélectionner la tête
- sélectionner la liste privée de la tête
- tester si une liste est vide
- on ne pourra pas demander la tête ou la queue d'une liste vide.
- une liste ayant une tête n'est pas vide
- si on a ajouté un élément à une liste, sa tête est cet élément et la queue est la liste initiale

- **Sorte :** Liste (liste d'objets de sorte E)
- Opérations :
  - vide : : Liste
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (observateur)
  - queue : : Liste → Liste (observateur)
- **Propriétés :**
  - tête(liste(x, xs)) = x
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)

- queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)

- queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool (testeur)
  - tête(liste(x, xs)) = x
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool (testeur)

- queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool (testeur)
- **Préconditions :**
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool (testeur)
- **Préconditions :**
  - tête(liste(x, xs)) = x
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool (testeur)
- **Préconditions :**
  - **pré** tête(xs) = ¬ est_vide(xs)
  - **pré** queue(xs) = ¬ est_vide(xs)
- Axiomes :
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool (testeur)
- **Préconditions :**
  - **pré** tête(xs) = ¬ est_vide(xs)
  - **pré** queue(xs) = ¬ est_vide(xs)
- **Axiomes :**
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool (testeur)
- **Préconditions :**
  - **pré** tête(xs) = ¬ est_vide(xs)
  - **pré** queue(xs) = ¬ est_vide(xs)
- **Axiomes :**
  - est_vide(vide) = Vrai
  - est_vide(liste(x, xs)) = Faux
  - tête(liste(x, xs)) = x
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool (testeur)
- **Préconditions :**
  - **pré** tête(xs) = ¬ est_vide(xs)
  - **pré** queue(xs) = ¬ est_vide(xs)
- **Axiomes :**
  - est_vide(vide) = Vrai
  - est_vide(liste(x, xs)) = Faux
  - tête(liste(x, xs)) = x
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool (testeur)
- **Préconditions :**
  - **pré** tête(xs) = ¬ est_vide(xs)
  - **pré** queue(xs) = ¬ est_vide(xs)
- **Axiomes :**
  - est_vide(vide) = Vrai
  - est_vide(liste(x, xs)) = Faux
  - tête(liste(x, xs)) = x
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool (testeur)
- **Préconditions :**
  - **pré** tête(xs) = ¬ est_vide(xs)
  - **pré** queue(xs) = ¬ est_vide(xs)
- **Axiomes :**
  - est_vide(vide) = Vrai
  - est_vide(liste(x, xs)) = Faux
  - tête(liste(x, xs)) = x
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool (testeur)
- **Préconditions :**
  - **pré** tête(xs) = ¬ est_vide(xs)
  - **pré** queue(xs) = ¬ est_vide(xs)
- **Axiomes :**
  - est_vide(vide) = Vrai
  - est_vide(liste(x, xs)) = Faux
  - tête(liste(x, xs)) = x
  - queue(liste(x, xs)) = xs

- **Sorte :** Liste (liste d'objets de sorte E)
- **Opérations :**
  - vide : : Liste (constructeur)
  - liste : : E Liste → Liste (constructeur)
  - tête : : Liste → E (sélecteur)
  - queue : : Liste → Liste (sélecteur)
  - est_vide : : Liste → Bool (testeur)
- **Préconditions :**
  - **pré** tête(xs) = ¬ est_vide(xs)
  - **pré** queue(xs) = ¬ est_vide(xs)
- **Axiomes :**
  - est_vide(vide) = Vrai
  - est_vide(liste(x, xs)) = Faux
  - tête(liste(x, xs)) = x
  - queue(liste(x, xs)) = xs

Un premier essai sur Python avec des tuples

```python
vide = None

def liste( x, xs ) :
    return ( x, xs )

def est_vide( xs ) :
    return xs == vide # ou pythonerie : return not xs

def tete( xs ) :
    assert ( not est_vide( xs ) ) , "Une liste vide n'a pas de tête"
    ( tete, _ ) = xs
    return tete       # ou pythonerie : xs[0]

def queue( xs ) :
    assert ( not est_vide( xs ) ) , "Une liste vide n'a pas de queue"
    ( _, queue ) = xs
    return queue      # ou pythonerie : xs[1:]
```

Un deuxième essai sur Python avec des tableaux

```python
vide = []

def liste( x, xs ) :
    return [x] + xs

def est_vide( xs ) :
    return xs == vide

def tete( xs ) :
    return xs[0]

def queue( xs ) :
    return xs[1:]
```

Longueur d'une liste

Exercice 1

En utilisant uniquement **`liste`**, **`est_vide`**, **`tete`**, **`queue`**, *quelque soit le moteur qui les implémente, déterminez une fonction* **`longueur(xs)`** *qui calcule la longueur de la liste* **`xs`** *donnée en argument.*

Longueur d'une liste

```
from liste_implem_list import *
#from liste_implem_tuple import *

##################################################
#                                                #
#  B a r r i è r e   d ' a b s t r a c t i o n   #
#                                                #
##################################################

l1 = liste(1, liste(2, liste(3, liste(4, vide))))

def longueur( xs ) :
    long_actuelle  = 0
    reste_a_lire = xs
    while not est_vide( reste_a_lire ) :
        long_actuelle += 1
        reste_a_lire = queue( reste_a_lire )
    return long_actuelle
```

Longueur d'une liste

```
from liste_implem_list import *
#from liste_implem_tuple import *

##################################################
#                                                #
#  B a r r i è r e   d ' a b s t r a c t i o n   #
#                                                #
##################################################

l1 = liste(1, liste(2, liste(3, liste(4, vide))))

def longueur( xs ) :
    if est_vide( xs ) :
        return 0
    return 1 + longueur( queue( xs ) )
```

Bien sûr on peut faire bien mieux avec un peu de programmation orientée objet mais cela dépasse le niveau de notre programme...

Avec OCaml

```
type 'a liste =
  |Vide
  |L of ('a * 'a liste);;

let ajoute el els = L(el,els);;

exception ListeVide;;

let tete = function
  |Vide -> raise ListeVide
  |L(el,els) -> el;;

let queue = function
  |Vide -> raise ListeVide
  |L(el,els) -> els;;
```

Il s'agit d'implémenter du bon côté de la barrière les fonctions définies ainsi dans la documentation d'un autre langage de programmation :

Exercice 2

```
last :: [a] -> a
Extract the last element of a list
```

Exercice 3

```
init :: [a] -> [a]
Return all the elements of a list except the last one
```

Exercice 4

Append two lists, i.e.,

```
[x1, ..., xm] ++ [y1, ..., yn] == [x1, ..., xm, y1, ..., yn]
[x1, ..., xm] ++ [y1, ...] == [x1, ..., xm, y1, ...]
```

Exercice 5

```
reverse :: [a] -> [a]
reverse xs returns the elements of xs in reverse order.
```

Exercice 6

```
sum :: Num a => [a] -> a
The sum function computes the sum of a finite list of numbers.

product :: Num a => [a] -> a
The product function computes the product of a finite list of numbers.
```

Exercice 7

```
any :: (a -> Bool) -> [a] -> Bool
Applied to a predicate and a list, any determines if any element of the list
        ↪   satisfies the predicate.

all :: (a -> Bool) -> [a] -> Bool
Applied to a predicate and a list, all determines if all elements of the list
        ↪   satisfy the predicate.
```

Exercice 8

```
maximum :: Ord a => [a] -> a
maximum returns the maximum value from a list, which must be non-empty, finite,
        ↪   and of an ordered type.

minimum :: Ord a => [a] -> a
minimum returns the minimum value from a list, which must be non-empty, finite,
        ↪   and of an ordered type.
```

Exercice 9

```
 takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile, applied to a predicate p and a list xs, returns the longest prefix
        ↪  (possibly empty) of xs of elements that satisfy p:
 takeWhile (< 3) [1,2,3,4,1,2,3,4] == [1,2]
 takeWhile (< 9) [1,2,3] == [1,2,3]
 takeWhile (< 0) [1,2,3] == []
```

Exercice 10

```
elem :: Eq a => a -> [a] -> Bool
elem is the list membership predicate
```

Exercice 11

```
find :: (a -> Bool) -> [a] -> Maybe a
The find function takes a predicate and a list and returns the first element in
        ↪ the list matching the predicate, or Nothing if there is no such
        ↪ element.
```

Exercice 12

```
filter :: (a -> Bool) -> [a] -> [a]
filter, applied to a predicate and a list, returns the list of those elements
        ↪  that satisfy the predicate; i.e.,
 filter p xs = [ x | x <- xs, p x]
```

Exercice 13

```
partition :: (a -> Bool) -> [a] -> ([a], [a])
The partition function takes a predicate a list and returns the pair of lists of
        ↪   elements which do and do not satisfy the predicate, respectively
```

Exercice 14

```
zip :: [a] -> [b] -> [(a, b)]
zip takes two lists and returns a list of corresponding pairs. If one input list
        ↪   is short, excess elements of the longer list are discarded.
```