STR1 : listes, files, piles

loig.jezequel@univ-nantes.fr

Qu'est-ce qu'une structure de données ?

Structure de données

Moyen pour représenter des données et leurs relations en mémoire et y accéder.

Pourquoi ne pas tout mettre dans des tableaux ?

Raisons de performances :

- L'ajout d'une donnée dans un tableau peut être coûteuse (recherche puis déplacement),
- le retrait d'une donnée d'un tableau peut être coûteux (recherche puis déplacement),

et aussi raisons pratiques : certaines relations entre données ne peuvent pas être représentées par les tableaux.

En pratique

Définition d'une structure de données

On décrit une interface : l'ensemble des opérations qu'on doit pouvoir réaliser sur la structure.

Exemple : interface du tableau (en Go)

- t[i] : accès au i-ème élément
- len(t) : accès au nombre d'éléments
- cap(t) : accès au nombre maximum d'éléments
- append(t, v) : ajout d'un élément à la fin

Implantation d'une structure de données

Il existe plusieurs façons de représenter (coder) une interface donnée, qui peuvent être plus ou moins efficaces.

Les listes

Principe

Une liste ℓ est une séquence d'éléments dont l'ordre est fixé.

Interface

- head(ℓ) : premier élément de la liste
- tail(ℓ) : liste des éléments sauf le premier
- append(v, ℓ) : liste constituée de v suivi de ℓ
- nil : liste vide
- isEmpty(ℓ) : vrai si ℓ est vide, faux sinon

Exemple d'intérêt par rapport aux tableaux

Tri par insertion : on évite la recopie des données après insertion d'un élément dans un ensemble trié.

Mise en œuvre des listes : listes chaînées

Élément de liste

Contenu :

- une valeur val, et
- (un pointeur vers) une liste next.

Liste représentée par son premier élément.

Lien avec l'interface

- head(ℓ) : $\ell.v$
- tail(ℓ) : $\ell.next$
- append(v, ℓ) : $val = v$, $next = \ell$
- nil : nil
- isEmpty(ℓ) : $\ell == nil$
- Pas dans l'interface, mais en bonus : next peut être utile pour parcourir la liste sans la modifier

Mise en œuvre des listes : listes chaînées

Élément de liste

Contenu :

- une valeur val, et
- (un pointeur vers) une liste next.

Liste représentée par son premier élément.

ℓ = append(1, nil)

Lien avec l'interface

- head(ℓ) : $\ell.v$
- tail(ℓ) : $\ell.next$
- append(v, ℓ) : $val = v$, $next = \ell$
- nil : nil
- isEmpty(ℓ) : $\ell == nil$
- Pas dans l'interface, mais en bonus : next peut être utile pour parcourir la liste sans la modifier

v = 1
next = nil

Mise en œuvre des listes : listes chaînées

Élément de liste

Contenu :

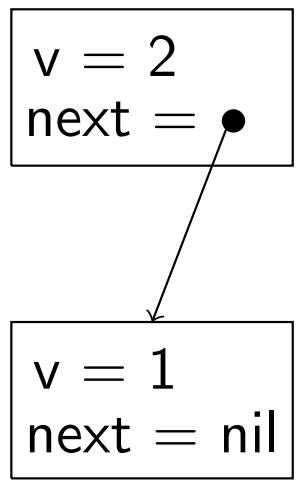
- une valeur val, et
- (un pointeur vers) une liste next.

Liste représentée par son premier élément.

$\ell = \text{append}(2, \ell)$

Lien avec l'interface

- head(ℓ) : $\ell.v$
- tail(ℓ) : $\ell.next$
- append(v, ℓ) : $val = v$, $next = \ell$
- nil : nil
- isEmpty(ℓ) : $\ell == nil$
- Pas dans l'interface, mais en bonus : next peut être utile pour parcourir la liste sans la modifier



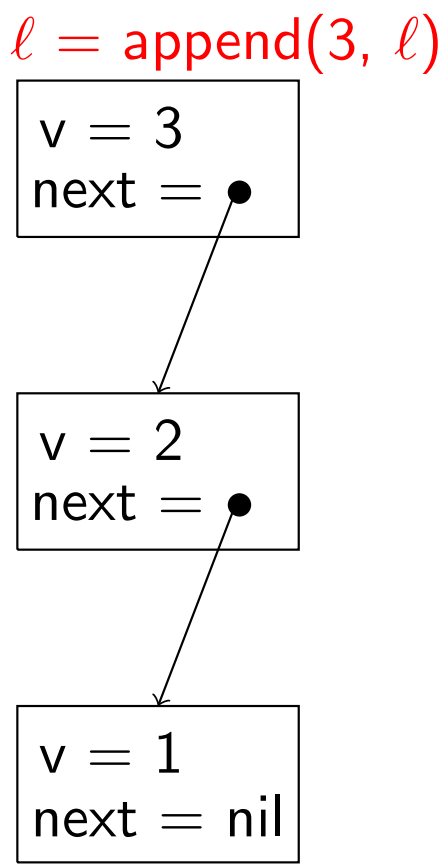
Mise en œuvre des listes : listes chaînées

Élément de liste

Contenu :

- une valeur val, et
- (un pointeur vers) une liste next.

Liste représentée par son premier élément.



Lien avec l'interface

- head(ℓ) : $\ell.v$
- tail(ℓ) : $\ell.next$
- append(v, ℓ) : $val = v$, $next = \ell$
- nil : nil
- isEmpty(ℓ) : $\ell == nil$
- Pas dans l'interface, mais en bonus : next peut être utile pour parcourir la liste sans la modifier

Mise en œuvre des listes : listes chaînées

Élément de liste

Contenu :

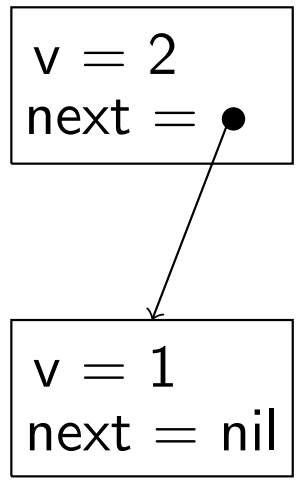
- une valeur val, et
- (un pointeur vers) une liste next.

Liste représentée par son premier élément.

$\ell = \text{tail}(\ell)$

Lien avec l'interface

- head(ℓ) : $\ell.v$
- tail(ℓ) : $\ell.next$
- append(v, ℓ) : $val = v$, $next = \ell$
- nil : nil
- isEmpty(ℓ) : $\ell == nil$
- Pas dans l'interface, mais en bonus : next peut être utile pour parcourir la liste sans la modifier



Tri par insertion sur les listes

Entrée

Une liste ℓ d'entiers.

Sortie

Une liste ℓ' contenant les mêmes entiers rangés du plus petit au plus grand.

parcours(ℓ, ℓ')

si isEmpty(ℓ),
alors retourner ℓ',
sinon soit $\ell' =$ insertion(head(ℓ), ℓ')
puis retourner parcours(tail(ℓ), ℓ').

insertion(v, ℓ)

si isEmpty(ℓ), alors retourner append(v, nil).
si head(ℓ) $\geq$ v,
alors retourner append(head(ℓ), append(v, tail(ℓ))),
sinon soit fin $=$ insertion(v, tail(ℓ))
puis retourner append(head(ℓ), fin).

Tri par insertion sur les listes, exemple

parcours([12, 32, 7, 23, 9], nil)

Tri par insertion sur les listes, exemple

parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)

Tri par insertion sur les listes, exemple

parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]

Tri par insertion sur les listes, exemple

parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])

Tri par insertion sur les listes, exemple

parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])

Tri par insertion sur les listes, exemple

parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)

Tri par insertion sur les listes, exemple

parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
        insertion(12, nil)
                retourner [12]
        parcours([32, 7, 23, 9], [12])
                insertion(32, [12])
                        insertion(32, nil)
                                retourner [32]
                        retourner [12, 32]
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
        insertion(12, nil)
                retourner [12]
        parcours([32, 7, 23, 9], [12])
                insertion(32, [12])
                        insertion(32, nil)
                                retourner [32]
                        retourner [12, 32]
                parcours([7, 23, 9], [12, 32])
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
        insertion(12, nil)
                retourner [12]
        parcours([32, 7, 23, 9], [12])
                insertion(32, [12])
                        insertion(32, nil)
                                retourner [32]
                        retourner [12, 32]
                parcours([7, 23, 9], [12, 32])
                        insertion(7, [12, 32])
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
        insertion(12, nil)
                retourner [12]
        parcours([32, 7, 23, 9], [12])
                insertion(32, [12])
                        insertion(32, nil)
                                retourner [32]
                        retourner [12, 32]
                parcours([7, 23, 9], [12, 32])
                        insertion(7, [12, 32])
                                retourner [7, 12, 32]
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
        insertion(12, nil)
                retourner [12]
        parcours([32, 7, 23, 9], [12])
                insertion(32, [12])
                        insertion(32, nil)
                                retourner [32]
                        retourner [12, 32]
                parcours([7, 23, 9], [12, 32])
                        insertion(7, [12, 32])
                                retourner [7, 12, 32]
                        parcours([23, 9], [7, 12, 32])
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
        insertion(12, nil)
                retourner [12]
        parcours([32, 7, 23, 9], [12])
                insertion(32, [12])
                        insertion(32, nil)
                                retourner [32]
                        retourner [12, 32]
                parcours([7, 23, 9], [12, 32])
                        insertion(7, [12, 32])
                                retourner [7, 12, 32]
                        parcours([23, 9], [7, 12, 32])
                                insertion(23, [7, 12, 32])
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
        insertion(12, nil)
                retourner [12]
        parcours([32, 7, 23, 9], [12])
                insertion(32, [12])
                        insertion(32, nil)
                                retourner [32]
                        retourner [12, 32]
                parcours([7, 23, 9], [12, 32])
                        insertion(7, [12, 32])
                                retourner [7, 12, 32]
                        parcours([23, 9], [7, 12, 32])
                                insertion(23, [7, 12, 32])
                                        insertion(23, [12, 32])
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
                            retourner [23, 32]
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
                            retourner [23, 32]
                        retourner [12, 23, 32]
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
                            retourner [23, 32]
                        retourner [12, 23, 32]
                    retourner [7, 12, 23, 32]
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
        insertion(12, nil)
                retourner [12]
        parcours([32, 7, 23, 9], [12])
                insertion(32, [12])
                        insertion(32, nil)
                                retourner [32]
                        retourner [12, 32]
                parcours([7, 23, 9], [12, 32])
                        insertion(7, [12, 32])
                                retourner [7, 12, 32]
                        parcours([23, 9], [7, 12, 32])
                                insertion(23, [7, 12, 32])
                                        insertion(23, [12, 32])
                                                insertion(23, [32])
                                                        retourner [23, 32]
                                                retourner [12, 23, 32]
                                        retourner [7, 12, 23, 32]
                                parcours([9], [7, 12, 23, 32])
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
                            retourner [23, 32]
                        retourner [12, 23, 32]
                    retourner [7, 12, 23, 32]
                parcours([9], [7, 12, 23, 32])
                    insertion(9, [7, 12, 23, 32])
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
                            retourner [23, 32]
                        retourner [12, 23, 32]
                    retourner [7, 12, 23, 32]
                parcours([9], [7, 12, 23, 32])
                    insertion(9, [7, 12, 23, 32])
                        insertion(9, [12, 23, 32])
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
                            retourner [23, 32]
                        retourner [12, 23, 32]
                    retourner [7, 12, 23, 32]
                parcours([9], [7, 12, 23, 32])
                    insertion(9, [7, 12, 23, 32])
                        insertion(9, [12, 23, 32])
                            retourner [9, 12, 23, 32]
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
                            retourner [23, 32]
                        retourner [12, 23, 32]
                    retourner [7, 12, 23, 32]
                parcours([9], [7, 12, 23, 32])
                    insertion(9, [7, 12, 23, 32])
                        insertion(9, [12, 23, 32])
                            retourner [9, 12, 23, 32]
                        retourner [7, 9, 12, 23, 32]
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
                            retourner [23, 32]
                        retourner [12, 23, 32]
                    retourner [7, 12, 23, 32]
                parcours([9], [7, 12, 23, 32])
                    insertion(9, [7, 12, 23, 32])
                        insertion(9, [12, 23, 32])
                            retourner [9, 12, 23, 32]
                        retourner [7, 9, 12, 23, 32]
                    retourner [7, 9, 12, 23, 32]
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
                            retourner [23, 32]
                        retourner [12, 23, 32]
                    retourner [7, 12, 23, 32]
                parcours([9], [7, 12, 23, 32])
                    insertion(9, [7, 12, 23, 32])
                        insertion(9, [12, 23, 32])
                            retourner [9, 12, 23, 32]
                        retourner [7, 9, 12, 23, 32]
                    retourner [7, 9, 12, 23, 32]
                retourner [7, 9, 12, 23, 32]
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
                            retourner [23, 32]
                        retourner [12, 23, 32]
                    retourner [7, 12, 23, 32]
                parcours([9], [7, 12, 23, 32])
                    insertion(9, [7, 12, 23, 32])
                        insertion(9, [12, 23, 32])
                            retourner [9, 12, 23, 32]
                        retourner [7, 9, 12, 23, 32]
                    retourner [7, 9, 12, 23, 32]
                retourner [7, 9, 12, 23, 32]
            retourner [7, 9, 12, 23, 32]
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
                            retourner [23, 32]
                        retourner [12, 23, 32]
                    retourner [7, 12, 23, 32]
                parcours([9], [7, 12, 23, 32])
                    insertion(9, [7, 12, 23, 32])
                        insertion(9, [12, 23, 32])
                            retourner [9, 12, 23, 32]
                        retourner [7, 9, 12, 23, 32]
                    retourner [7, 9, 12, 23, 32]
                retourner [7, 9, 12, 23, 32]
            retourner [7, 9, 12, 23, 32]
```

Tri par insertion sur les listes, exemple

```
parcours([12, 32, 7, 23, 9], nil)
    insertion(12, nil)
        retourner [12]
    parcours([32, 7, 23, 9], [12])
        insertion(32, [12])
            insertion(32, nil)
                retourner [32]
            retourner [12, 32]
        parcours([7, 23, 9], [12, 32])
            insertion(7, [12, 32])
                retourner [7, 12, 32]
            parcours([23, 9], [7, 12, 32])
                insertion(23, [7, 12, 32])
                    insertion(23, [12, 32])
                        insertion(23, [32])
                            retourner [23, 32]
                        retourner [12, 23, 32]
                    retourner [7, 12, 23, 32]
                parcours([9], [7, 12, 23, 32])
                    insertion(9, [7, 12, 23, 32])
                        insertion(9, [12, 23, 32])
                            retourner [9, 12, 23, 32]
                        retourner [7, 9, 12, 23, 32]
                    retourner [7, 9, 12, 23, 32]
                retourner [7, 9, 12, 23, 32]
            retourner [7, 9, 12, 23, 32]
        retourner [7, 9, 12, 23, 32]
    retourner [7, 9, 12, 23, 32]
```

Mise en œuvre des listes : listes doublement chaînées

Élément de liste

Contenu :

- une valeur val, et
- (un pointeur vers) une liste next.
- (un pointeur vers) une liste prev.

Liste représentée par son premier élément.

Lien avec l'interface

- head(ℓ) : $\ell.v$
- tail(ℓ) : $\ell.next$
- append(v, ℓ) : $val = v$, $next = \ell$, $prev = nil$
- nil : nil
- isEmpty(ℓ) : $\ell == nil$
- Bonus : next et prev utiles pour parcourir la liste sans la modifier

Mise en œuvre des listes : listes doublement chaînées

Élément de liste

Contenu :

- une valeur val, et
- (un pointeur vers) une liste next.
- (un pointeur vers) une liste prev.

Liste représentée par son premier élément.

ℓ = append(1, nil)

Lien avec l'interface

- head(ℓ) : $\ell.v$
- tail(ℓ) : $\ell.next$
- append(v, ℓ) : $val = v$, $next = \ell$, $prev = nil$
- nil : nil
- isEmpty(ℓ) : $\ell == nil$
- Bonus : next et prev utiles pour parcourir la liste sans la modifier

v = 1
pred = nil
next = nil

Mise en œuvre des listes : listes doublement chaînées

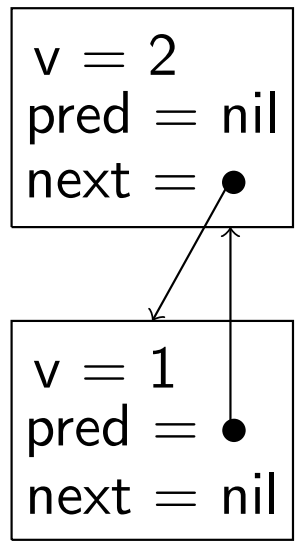
Élément de liste

Contenu :

- une valeur val, et
- (un pointeur vers) une liste next.
- (un pointeur vers) une liste prev.

Liste représentée par son premier élément.

$\ell = \text{append}(2, \ell)$



Lien avec l'interface

- head(ℓ) : $\ell.v$
- tail(ℓ) : $\ell.next$
- append(v, ℓ) : $val = v$, $next = \ell$, $prev = nil$
- nil : nil
- isEmpty(ℓ) : $\ell == nil$
- Bonus : next et prev utiles pour parcourir la liste sans la modifier

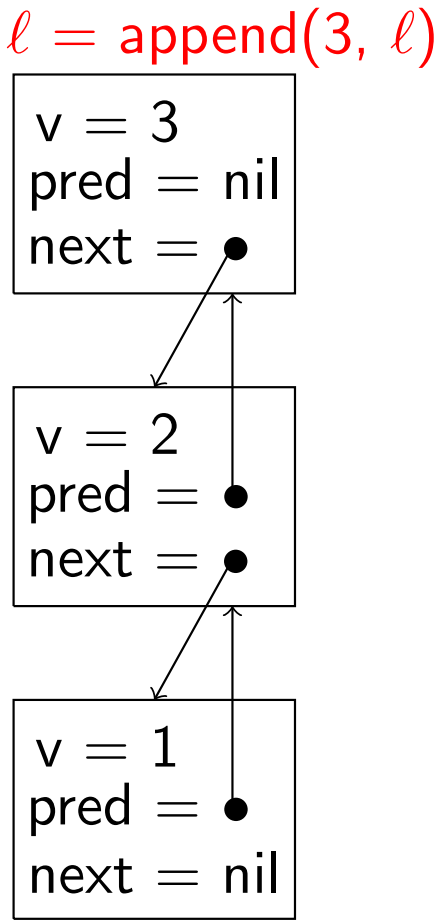
Mise en œuvre des listes : listes doublement chaînées

Élément de liste

Contenu :

- une valeur val, et
- (un pointeur vers) une liste next.
- (un pointeur vers) une liste prev.

Liste représentée par son premier élément.



Lien avec l'interface

- head(ℓ) : $\ell.v$
- tail(ℓ) : $\ell.next$
- append(v, ℓ) : $val = v$, $next = \ell$, $prev = nil$
- nil : nil
- isEmpty(ℓ) : $\ell == nil$
- Bonus : next et prev utiles pour parcourir la liste sans la modifier

Mise en œuvre des listes : listes doublement chaînées

Élément de liste

Contenu :

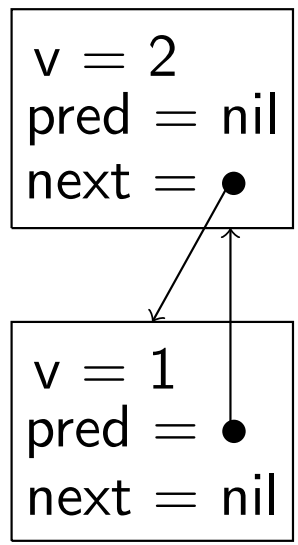
- une valeur val, et
- (un pointeur vers) une liste next.
- (un pointeur vers) une liste prev.

Liste représentée par son premier élément.

$\ell = \text{tail}(\ell)$

Lien avec l'interface

- head(ℓ) : $\ell.v$
- tail(ℓ) : $\ell.next$
- append(v, ℓ) : $val = v$, $next = \ell$, $prev = nil$
- nil : nil
- isEmpty(ℓ) : $\ell == nil$
- Bonus : next et prev utiles pour parcourir la liste sans la modifier



Piles et files

Ordre d'ajout des éléments dans un ensemble

Piles et files sont des structures de données qui permettent de se souvenir de l'ordre dans lequel des éléments ont été ajoutés à un ensemble.

Pile. Privilégie le dernier élément ajouté

File. Privilégie le premier élément ajouté

Exemples d'utilisations

Parcours de structures complexes (graphes), stockage d'opérations à réaliser dans un certain ordre, backtracking, etc.

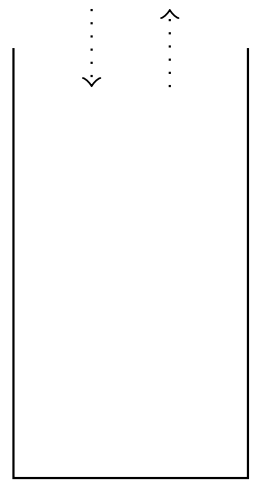
Les piles

Principe

LIFO, Last In First Out, le dernier élément ajouté est le premier à être récupéré lors d'un accès à une pile.

Interface

- push(v, P) : ajoute un élément v sur la pile P
- pop(P) : retire le sommet de la pile P et indique sa valeur
- nil : pile vide
- isEmpty(P) : vrai si la pile P est vide, faux sinon



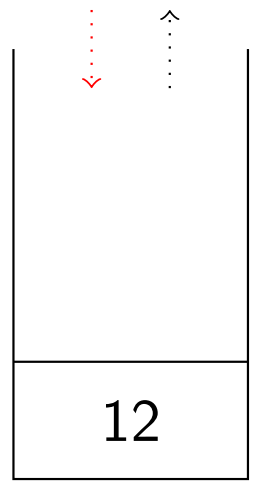
Les piles

Principe

LIFO, Last In First Out, le dernier élément ajouté est le premier à être récupéré lors d'un accès à une pile.

Interface

- push(v, P) : ajoute un élément v sur la pile P
- pop(P) : retire le sommet de la pile P et indique sa valeur
- nil : pile vide
- isEmpty(P) : vrai si la pile P est vide, faux sinon



push(12, P)

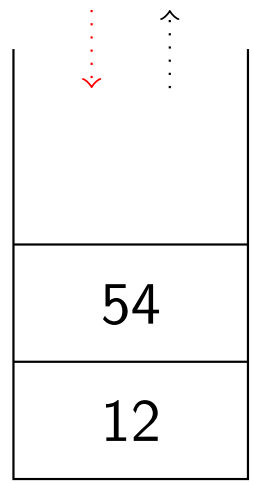
Les piles

Principe

LIFO, Last In First Out, le dernier élément ajouté est le premier à être récupéré lors d'un accès à une pile.

Interface

- push(v, P) : ajoute un élément v sur la pile P
- pop(P) : retire le sommet de la pile P et indique sa valeur
- nil : pile vide
- isEmpty(P) : vrai si la pile P est vide, faux sinon



push(54, P)

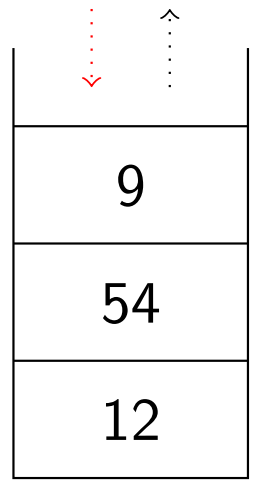
Les piles

Principe

LIFO, Last In First Out, le dernier élément ajouté est le premier à être récupéré lors d'un accès à une pile.

Interface

- push(v, P) : ajoute un élément v sur la pile P
- pop(P) : retire le sommet de la pile P et indique sa valeur
- nil : pile vide
- isEmpty(P) : vrai si la pile P est vide, faux sinon



push(9, P)

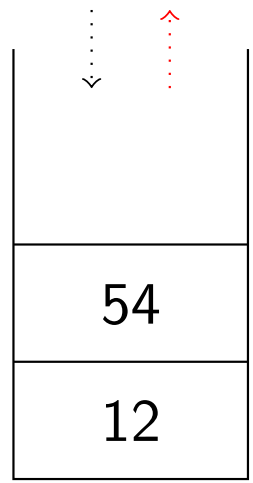
Les piles

Principe

LIFO, Last In First Out, le dernier élément ajouté est le premier à être récupéré lors d'un accès à une pile.

Interface

- push(v, P) : ajoute un élément v sur la pile P
- pop(P) : retire le sommet de la pile P et indique sa valeur
- nil : pile vide
- isEmpty(P) : vrai si la pile P est vide, faux sinon



pop(P)

Les piles : implantation

Tableaux

- push : ajout d'un élément en fin de tableau
- pop : récupération du dernier élément du tableau et réduction de la taille du tableau de 1
- nil : tableau vide
- isEmpty : test si la longueur du tableau est 0

Listes

On peut coder les piles à l'aide des listes, ou en adaptant l'implantation des listes à base de pointeurs :

- push : append
- pop : head modifiée pour supprimer l'élément
- nil : nil
- isEmpty : isEmpty

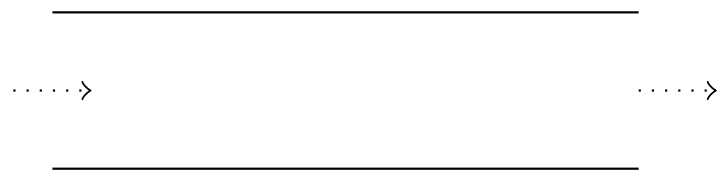
Les files

Principe

FIFO, First In First Out, le premier élément ajouté est le premier à être récupéré lors d'un accès à une file.

Interface

- push(v, F) : ajoute un élément v sur la file F
- pull(F) : retire le premier élément de la file F et indique sa valeur
- nil : file vide
- isEmpty(F) : vrai si la file F est vide, faux sinon



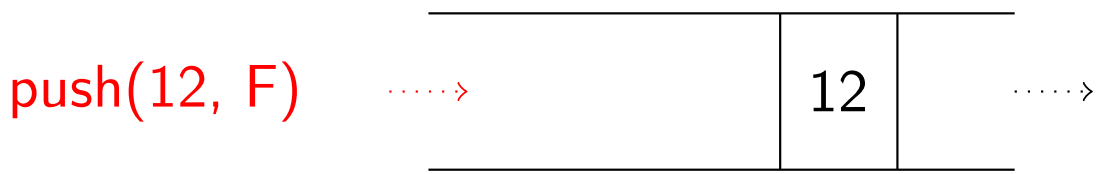
Les files

Principe

FIFO, First In First Out, le premier élément ajouté est le premier à être récupéré lors d'un accès à une file.

Interface

- push(v, F) : ajoute un élément v sur la file F
- pull(F) : retire le premier élément de la file F et indique sa valeur
- nil : file vide
- isEmpty(F) : vrai si la file F est vide, faux sinon



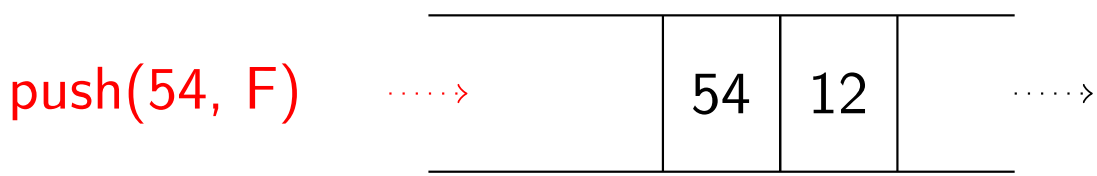
Les files

Principe

FIFO, First In First Out, le premier élément ajouté est le premier à être récupéré lors d'un accès à une file.

Interface

- push(v, F) : ajoute un élément v sur la file F
- pull(F) : retire le premier élément de la file F et indique sa valeur
- nil : file vide
- isEmpty(F) : vrai si la file F est vide, faux sinon



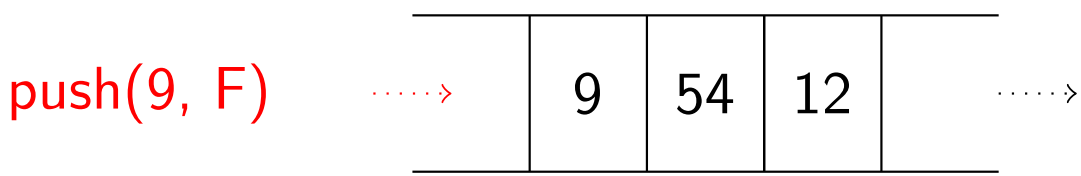
Les files

Principe

FIFO, First In First Out, le premier élément ajouté est le premier à être récupéré lors d'un accès à une file.

Interface

- push(v, F) : ajoute un élément v sur la file F
- pull(F) : retire le premier élément de la file F et indique sa valeur
- nil : file vide
- isEmpty(F) : vrai si la file F est vide, faux sinon



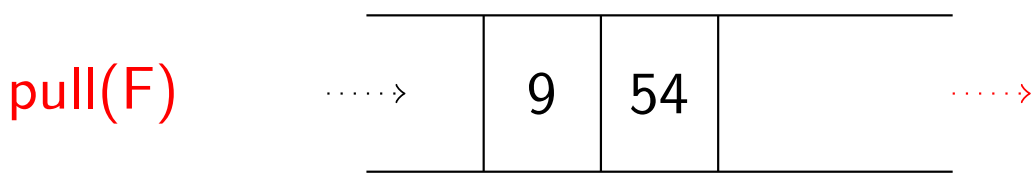
Les files

Principe

FIFO, First In First Out, le premier élément ajouté est le premier à être récupéré lors d'un accès à une file.

Interface

- push(v, F) : ajoute un élément v sur la file F
- pull(F) : retire le premier élément de la file F et indique sa valeur
- nil : file vide
- isEmpty(F) : vrai si la file F est vide, faux sinon



Les files : implantation

Tableaux

- push : ajout d'un élément en fin de tableau
- pull : récupération du premier élément du tableau et réduction de la taille du tableau de 1
- nil : tableau vide
- isEmpty : test si la longueur du tableau est 0

Listes

On peut coder les piles à l'aide des listes, ou en adaptant l'implantation des listes à base de pointeurs :

- push : append
- pull : récupérer le dernier élément et le supprimer
- nil : nil
- isEmpty : isEmpty

Un pointeur vers la fin de la liste améliore les performances.